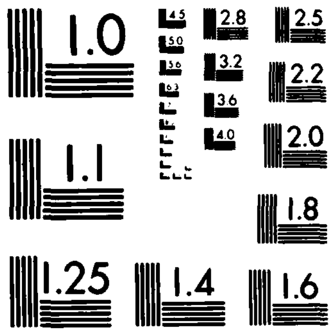
AD-A151 313   THE DESIGN AND IMPLEMENTATION OF A SYNTAX DIRECTED   1/2
EDITOR FOR A SPACE CONSTRAINED MICROCOMPUTER(U) NAVAL
POSTGRADUATE SCHOOL MONTEREY CA R F RICHBOURG JUN 84

UNCLASSIFIED   F/G 9/2   NL


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

(2)

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

THE DESIGN AND IMPLEMENTATION OF A SYNTAX DIRECTED EDITOR FOR A SPACE CONSTRAINED MICROCOMPUTER

by

Robert F. Richbourg

June 1984

Thesis Advisor: Bruce J. MacLennan

Approved for public release; distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A151 373	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The Design and Implementation of a SYNTAX Directed Editor for a Space Constrained Microcomputer		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June 1984
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Robert F. Richbourg		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		12. REPORT DATE June 1984
		13. NUMBER OF PAGES 100
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Syntax directed editor, programming environments, c programming language, design

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Syntax directed editors (SDE) have been built to support popular languages or subsets of those languages. Typically, these implementations require large amounts of computing resources. This work describes the design and implemtnation of a SDE which requires less than 58 thousand bytes of main memory and supports the full C programming language. Several extant SDE models are examined in an effort to define a basic set of SDE (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT (Continued)

facilities. Design principles are combined with machine constraints to produce a plan for the implementation of these facilities. A sample session with the resulting editor is provided.

The syntactic irregularities of the C programming language are examined. A discussion showing how language irregularities can hamper the implementation of a SDE follows. A grammatical definition of the C language is included.

ssion For		
[illegible]	✓	
TAB	☐	
nounced	☐	
tion		
PER CALL JC		
ibution/		
ilability Codes		
Dist	Avail and/or Special	
A-1	[illegible]	

The Design and Implementation of a Syntax Directed Editor for a Space Constrained Microcomputer

by

Robert F. Richbourg
Captain, United States Army
B. S., Wake Forest University, 1976

Submitted in partial fullfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June, 1984

Author: Robert F. Richbourg

Approved by: [illegible]
Thesis Advisor

[illegible]
Second Reader

[illegible]
Chairman, Department of Computer Science

[illegible]
Dean of Information and Policy Sciences

ABSTRACT

Syntax directed editors (SDE) have been built to support popular languages or subsets of those languages. Typically, these implementations require large amounts of computing resources. This work describes the design and implementation of a SDE which requires less than 58 thousand bytes of main memory and supports the full C programming language. Several extant SDE models are examined in an effort to define a basic set of SDE facilities. Design principles are combined with machine constraints to produce a plan for the implementation of these facilities. A sample session with the resulting editor is provided.

The syntactic irregularities of the C programming language are examined. A discussion showing how language irregularities can hamper the implementation of a SDE follows. A grammatical definition of the C language is included.

Additional keywords: theses, C programming language, programming manuals

TABLE OF CONTENTS

I. INTRODUCTION

Editors have existed in various forms since the advent of the computer itself. The first editors were very primitive, non-interactive models used to manipulate unit records of card images. As computer usage, availability, applications, and power increased, the facilities provided by editors also grew. From the primitive unit record editors came the batch editors which allowed the insertion of a single card within a batch to make global changes to all cards in that batch. Then came the interactive line editors which greatly increased the power of the editing tool, but were still conceptually tied to the idea of an eighty column card image. Truncation of input was a serious problem with these editors. Variable length line editors arrived and partly solved the problem by employing the notion of a super line [Ref. 1], commonly set to a maximum capacity of five hundred characters. In such editors, the truncation problem was delayed, but not solved. With the arrival of stream editors, the truncation problem was finally solved and the link to the card image or unit record broken. These editors were still inconvenient models by modern standards. Full screen editors followed and became the basis for modern, interactive editing tools. These editors allowed any line on screen to be edited and, more importantly, provided for immediate screen updates of changes made to the file.

The full screen editor proved to be very convenient for the user. Also, the functions performed by such editors became very powerful. Global search and replace commands, block moves, file insertion, and similar features were commonly available. Generally, these editors required a second, follow-on program, called a formatter to produce the desired output format of the object being edited. The combination of these two programs into one was the logical extension of the full screen editor. The inclusion of the formatter into the editor was an attempt to provide another feature designed to help the user. Such utilities came to be known as structure editors because they were able to impose some type of structure on the object as it was being edited. The user was able to see the final form of his project while in the process of creating the project.

The ideas involved in structure editors gave rise to the concept of the language or syntax directed editor (SDE). These editors are primarily intended to support the editing of programs as opposed to text. They have a built in 'knowledge' of the syntactic rules of some language and can help the user by providing assistance with the grammatical conventions of that language. They also have the abilities of structure editors in that they produce programs in accordance with standard, structured programming conventions. They generally include all the features commonly found in modern full screen editors as well.

Syntax directed editors thus allow a user to create and modify a program in terms of the syntactic nature of the supported language. They are able to prevent many lexical errors and can, in effect, themselves produce much of the code needed by the average user. As the editor knows the syntax of the supported language, the user is normally freed from entering many of the basic syntactic units (the common elements of a FOR statement for example) found in most programs. Thus, fewer keystrokes and less time are required to create programs. Because of these features, the syntax directed editor can ensure that many simple errors are routinely avoided. Thus, it can be expected that programmer productivity will increase as syntax directed editors become available in greater numbers.

A problem with most of the extant SDE systems is that many support only a carefully selected subset of some chosen language and they generally require great amounts of computer resources for support. A logical goal then would be to produce a syntax directed editor which fully supports some language and can be hosted on a minimally resourced machine. One method to achieve this goal is to first study the extant SDE implementations in an effort to determine some set of basic features. Then, select a commonly available language as the target language. Finally, attempt to design and implement a syntax directed editor on a microcomputer. This approach represents the main thrust of the effort for this thesis work.

The first step is to examine existing SDE implementations. An effort will be made to extract a basic set of features common among the different models. The results of this examination will then be combined with a set of engineering principles. These two factors will provide guidelines for the design and implementation of a small scale or baseline model syntax directed editor.

II. EXISTING SDE SYSTEMS

There are many existing syntax directed editors. Most of them have been installed on computers supporting large numbers of users. Typically, the systems are academic or experimental in nature. This has an impact on the design philosophy behind each implementation. As an example, if the editor is intended to be used in an academic environment then the editor generally includes features which enforce the syntactic correctness of the program as it is being edited. Such editors may prevent the entry of any incorrect structure. Other models force the user to correct every error as it occurs. That is, if an error has occurred, no further entry of any type is allowed until that error is corrected. More lenient versions make note of any errors and display them so that the user's attention is drawn to them. Reverse video is often used for this purpose. Erroneous constructs will be displayed in this manner until they are corrected. The overhead and inconvenience that can be caused by systems that enforce correctness might not be viewed as worthwhile in a commercial environment.

Another result of the nature of the current implementations is that they are not very space efficient. Many use a tree structure to represent the object being edited because of the natural correlation between trees and hierarchical structures, such as programs. The tree structure itself and the necessary code to move about within the tree will consume a large amount of memory. There are alternatives to the tree representation which are not as convenient, but can result in memory savings.

A third characteristic of many SDE systems is that they are often designed to be a single component in an integrated programming environment. The editor may be designed to perform parsing and lexical analysis for the compiler. It may provide an immediate execution facility as an outgrowth of this architecture. As an alternative, the editor may be directly linked to or contain an interpreter which supports an immediate execution facility.

None of the characteristics discussed above are truly essential to a syntax directed editor. They do represent very convenient features and are included in most of the SDE implementations which will be discussed. However, in an effort to define the basic requirements of a SDE, the correctness issue, the internal data structure, and the immediate execution facility should be discounted. The 'correctness' of a program can be very dependent on the compiler which must ultimately generate executable code. The underlying data structure used by the editor should be transparent to the user. He should be totally unaware of internal representations of his program. Finally, immediate execution is a very pleasing feature. However, it is also very expensive to implement. In a basic SDE implementation, this expense is best left to a separate compiler.

With these assumptions in mind, several extant SDE implementations will be reviewed. Differences in the facilities they provide as well as their user interfaces will be noted. The models chosen for examination are representative of several design philosophies and span nearly fifteen years of SDE construction. Emily, MENTOR, Interlisp, the Cornell Program Synthesizer, Z, SED, and COPE will be discussed. With the exception of Interlisp, the discussions will be presented according to the chronological order of each system's inception.

A. EMILY

Emily [Ref. 2] represents one of the initial attempts to implement a syntax directed editor. It was designed by W. J. Hansen at Stanford University and was completed by 1971. The system is not a part of an integrated environment but is intended to support any language which can be described in terms of a Backus-Naur form (BNF) grammar. Emily accepts any BNF specification and allows the user to create a program by selecting productions from the specification. In a sense, Emily uses a template method of program creation down to the lowest level of the supported language's constructs. As an example, suppose that the grammar of Figure 1 represents a language supported by Emily. Emily keeps track of the

```
<stmt> ::= <var> = <expression> |
        BEGIN <stmt*> ; END |
        WHILE <expression> <stmt*> ;
<stmt*> ::= <stmt> |
        <stmt> <stmt*>
<expression> ::= (<expression>) |
        <expression> + <expression> |
        <var>
<var> ::= <identifier>
```

Fig. 1 Hypothetical BNF Specification

current node in the program development tree. If the current node was <stmt> then the user would be presented with a menu consisting of the three options for a <stmt> as listed in the grammar. He would select one of the options by pointing a light pen at the desired menu option. In this manner, an entire program would be created. Figure 2 shows the sequence of actions necessary to enter code under the Emily system. In Figure 2, the current node is underlined and the menu is listed underneath the generated code.

```
A.  <stmt>                                  B.  WHILE <expr>
                                                    <stmt*>


          <var> = <expr>                            (<expr>)
          BEGIN <stmt*> ; END                       <expr> + <expr>
          WHILE <expr> <stmt*>                      <var>


C.  WHILE <var>                             D.  WHILE TRUE
     <stmt*>                                        <stmt*>


          <identifier>                              <stmt>
                                                    <stmt> <stmt*>
```

Fig. 2 Programming By Selection

Emily insures that an incorrect program will never be entered. Every statement accepted has been derived from the grammar. As a result, there could be a very long sequence of derivations necessary to reach the lowest levels of the grammar. In the example of Figure 2, only two intermediate templates had to be selected in order to enter the identifier. However, this chain could be much longer and the resulting inconvenience much greater.

In addition to the grammar specification, Emily requires a second table which Hansen called the concrete syntax [Ref. 2, p. 60]. This table was used to produce the display characteristics of each grammar rule. As an example, indentation rules were created from this table. The two tables together naturally lead to the internal representation of the edited object as an abstract syntax tree.

Emily also supports holophrasting, a term defined by Hansen which implies that the program can be viewed from different levels. The details of the program can be suppressed and the user may inspect his code from a higher level. The level of detail can be set by the user. Thus, at a high level, only function names might be displayed. Once the user determined which function he wanted to examine in detail, he could reset the level and 'zoom in' on that function.

The tree representation has an impact on the methods a user must master in order to effect modifications to his program. The methods are very similar to those discussed in connection with the Cornell Program Synthesizer and will not be presented here. Another result of the abstract syntax tree representation is that the user must specify cursor movement commands in terms of the tree structure. These commands are very different from those found in conventional text editing systems. This is a trait shared by most SDE implementations which use a tree as the underlying data structure. The choice of using conventional text cursor commands as opposed to tree movement commands is currently a subject of some debate [Ref. 3]. The issues involved are based primarily on opinions and will not be presented.

B. MENTOR

MENTOR [Ref. 4] is a syntax directed editor which supports the Pascal language. The system was described by Veronique Donzeau-Gouge at the Workshop on Programming Languages held at Ridgefield, Ct. in June, 1980.

MENTOR is not part of an integrated system, nor does it support immediate execution of programs. A serious shortcoming of the system is that there is no interactive update of the screen to reflect the changes as they occur. Instead, a user must request screen refreshes as needed. This drawback is particularly inconvenient because of MENTOR's tree movement cursor commands. Again, an explicit request to refresh the screen must be issued in order to ascertain the current cursor position.

MENTOR accepts input as text strings. Programs are thus entered in a conventional manner, a character at a time. As the text is entered, MENTOR parses it into an abstract syntax tree representation. When editing, a user views the program in its tree form. This is a slight inconsistency in that to enter a program, text commands are used, but to edit the same program, tree cursor movement commands must be used.

MENTOR utilizes MENTOL, a general tree manipulation language, to effect tree movement commands during editing. The MENTOL commands constitute a language unto themselves and require some learning effort on the part of the user. Some of the MENTOL commands can have a very strange appearance to the uninitiated. As an example, "@TXT F @ if $v1 then x:=$v2 else $v3" will look in a subtree denoted by the marker "@TXT" for the next if statement containing an assignment to the variable x in its then part.

The MENTOR implementors based their choice of a non-interactive screen on portability issues. The system uses teletype compatible output so that it can be transported to almost any machine. Their decision to utilize a tree structure while editing and a text structure for creation is based on their decision not to maintain two internal representations of the program (i.e. one as a tree and one as text). Rather, the program is unparsed and pretty-printed on the screen as needed. Holophrasting is available in MENTOR. Many design decisions reflect the engineering tradeoff of sacrificing user convenience to other considerations.

C. INTERLISP

Interlisp [Ref. 5] is a growing and constantly changing system that has been built over a period of years by several key people. The system is so well integrated that it is difficult to discuss only the features of the editor. Thus, a brief overview of the major facilities available in the Interlisp system is presented.

The Programmer's Assistant (PA) is the editing facility in Interlisp. One of the PA's most distinctive features is that it maintains a history list of the user's input, a description of the side effects of operations, and the results of operations. This is a powerful mechanism which allows use of the REDO and UNDO commands. REDO allows a user to repeat a particular operation or series of commands. UNDO negates changes and can restore a file to an earlier state. UNDO can also be used to execute different versions of a file.

Because of the highly integrated nature of Interlisp, a user of the PA also has access to the Do-What-I-Mean (DWIM) and Masterscope facilities. DWIM is a very powerful facility which, in the case of incorrect input, can make a good guess as to the user's actual intent and attempt to execute the intended operation. Correction of spelling errors is a good example of a DWIM capability. Masterscope is a facility which can cross-reference user programs to determine where variables are declared, functions are called, and similar actions occur. It is intended to assist the user in locating side effects created by changes to low level utilities in a large program or set of programs. It maintains a database of its findings for each user and can respond to such queries as "Who uses FOO freely".

The structure of Lisp lends itself to the tree representation used by Interlisp. Programs are entered sequentially as characters but are edited as tree structures. This difference is not important in Lisp because of the very simple syntax of the language.

Many features in modern SDE systems were first implemented in Interlisp. Lisp, by nature of its structure, lends itself to many of the facilities found in Interlisp. These facilities do not often carry over easily to block structured languages. However, Interlisp is an important system that generally precedes block structured editors in terms of facilities provided.

D. CORNELL PROGRAM SYNTHESIZER

The Cornell Program Synthesizer (CPS) [Ref. 6] is an integrated environment which was available as early as 1981 and is generally attributed to Tim Teitelbaum of Cornell. A SDE is the heart of the system. Originally, CPS supported PL/CS, a subset of PL/I. This SDE is a table driven model however, and later versions of CPS support Pascal as well [Ref. 1].

CPS features a hybrid type editor. Programs are represented internally as tree structures. Program creation is accomplished both by direct entry and by selection. Templates are used in manner very similar to the Emily implementation. However, these templates are available only for the higher level constructs of the supported language. Low level components, such as identifiers and expressions are entered directly. Correspondingly, cursor movement is performed differently at the two levels. Tree movement is the rule when editing high level components. Character by character movement is performed at the low levels.

Editing programs under CPS is also accomplished at two levels. A user may not directly edit any template. These may only be deleted or inserted. In fact, the cursor is never allowed to rest directly on a template when in the edit mode. Low level expressions, however, may be freely edited. These are parsed on input and placed directly into a tree structure. Modifications under this scheme can be more cumbersome than when a conventional text editor is being used. As an example, suppose that a program contains the PL/I construct of Figure 3. If a user needs to place this construct within a while loop, a set

```
IF (INDEX = LIMIT)
      THEN PUT SKIP LIST('LIMIT REACHED')
```

Fig. 3 PL/I Program Construct

procedure similar to the modification methods used under Emily must be followed. First, the IF template must be 'clipped' (a temporary deletion) from the program to be stored in a temporary buffer. Then, a WHILE template would be inserted at the original position of the IF statement. The cursor is then moved to the statement placeholder within the WHILE template. At this point, the IF statement is reinserted into the program. The procedure in this simple example is not complicated. However, if many lines of code must be moved, the technique can be inconvenient. To relocate code within an existing loop is more difficult. When using a conventional editor, one would simply move the 'END' statement.

In CPS, the 'BEGIN' and 'END' are part of the same template, so neither can be moved independantly. Thus, the code to be included would have to be clipped, the 'BEGIN', 'END' pair would have to be expanded by inserting a statement template, and then the code could be inserted. This procedure is more inconvenient than that which would be required by a conventional text editor. The tradeoff is that CPS attempts to insure that any program being edited is always maintained syntactically correct. In the event that correctness can not be insured, constructs in error are highlighted in reverse video.

A major contribution of CPS is that it was one of the first implementations of a SDE for a block structured language to be a part of a total environment. The system features an incremental compilation scheme which allows immediate execution of programs being edited. CPS also improved user convenience as opposed to earlier implementations. A drawback to the system is that dual modes are required to accomplish modifications. CPS has not been able to match the editing convenience available in conventional text editors.

Another less than desirable feature of CPS concerns the commenting conventions. CPS provides a comment template which contains a statement template as an integral part. Thus, each comment is related to a statement or block of statements, and the comment appears in a fixed position above the statement it refers to. This convention does lend regularity to the appearance of comments. However, the user is restricted in that he may not place a comment on the same line with a program statement. The convention was necessary to deal with the arbitrary nature of comment placement. An editor is not allowed the compiler's luxury of simply throwing comments away. They must be saved and recognized in their own right. Further, if comments are allowed in a totally arbitrary fashion, it is not possible to produce a comment template easily.

E. Z

Z is a structure editor developed at Yale University which can operate on program structures, given a grammatical description of the target language. It has been called the "95% program editor" [Ref. 7] because it can accomplish approximately 95% of the functions that the implementors felt a SDE should be able to provide. Z is capable of editing any hierarchically structured object, given a description of that object. It has been used to support several programming languages including Lisp, APL, Pascal, and Bliss.

Z treats the objects it edits as text. A tree representation is not used in any fashion. Thus, all textual cursor movement commands are valid and every component of a created object is inserted sequentially, character by character. Z imposes structure (indentation for program objects) on its objects by means of a predefined table of information. This is similar to Emily's concrete syntax table. As an example, Z's table might contain information pertaining to the recognition of a Begin – End block which would allow the editor to insure the correct indentation of the statements within that block. Z also 'knows' how to insure balanced parenthesis, quotation marks, and similar context free constructs. The editor is also able to support holophrasting by using the indentation level of each statement. Z is not part of an integrated environment. It is linked to a compiler in that the editor may be temporarily suspended while a compilation is performed. Any error messages generated by the compiler are recognized by Z when it regains control. A compiler is not integrated directly into Z because the implementors felt that "the programmer is the person best able to decide when his program is in a state ready for compilation" and further, that "existing compilers are perfectly able to locate errors" [Ref. 7, p. 5].

Thus, Z is a very convenient system to use. It features straight-forward textual cursor movement commands at every level and can help the user to ensure that simple things are done correctly. It does not attempt to parse user input. Therefore, program correctness can not be ensured. It is a full screen editor and does not suffer the limitations of the MENTOR teletype approach. Z is also available as a general document editor. It can perform spelling checking and general word processing tasks when used in the document mode. Not all of these features are available in the program editing mode.

F. SED

SED (Syntax Editor) is a SDE implementation described by Lloyd Allison of the University of Western Australia in 1983 [Ref. 8]. The system supports the Pascal language and is hosted on a PDP 11/60 computer. SED is not part of an integrated environment and is similar to the Z system in the design philosophy governing the facilities that a SDE should include.

A central idea of the SED system is that "the editor should edit and that pretty-printing, type checking, and so on should be performed by optional commands or other programs" [Ref. 8, p. 454]. SED does not force any absolute standards on users. Even pretty-printing is an option which, when used, provides suggestions, not absolute standards. Similarly, SED does not attempt to ensure program correctness. An effort is made to parse user input "as well as possible". Allison has distinguished "long distance" and "short distance" syntax checking. Short distance refers to the area immediately surrounding a single statement. A long distance situation would occur, as an example, when a variable reference is made somewhere in a program and the declaration for that variable has been deleted. SED makes no attempt to correct or even identify long distance syntax errors.

This decision exemplifies SED's overall design philosophy. The user is held responsible for the ultimate correctness of his program. Problems such as the dangling else are ignored because of the idea that a compiler will locate such errors. A philosophy very similar to that of Z is apparent throughout the SED design.

SED does attempt to parse user input. As a result, an abstract syntax tree is used as an internal data structure. Templates are not available and all input is sequential. SED has chosen this implementation in an effort to reduce the storage requirements of the tree structures. Each individual token is not a separate node in the tree. Rather, strings containing pointers to substrings are used. The tree is collapsed to its textual form for off line storage. This requires the tree to be constructed each time an object is read in to be edited. SED also uses the tree to support cursor movement commands. Textual movement commands are generally not available.

Thus, SED is a less ambitious approach to the implementation of a syntax directed editor. It does offer greater functionality than Z in that SED attempts to ensure a degree of program correctness. However, the ultimate arbiter of correctness is a separate compiler.

4. COPE

COPE (a Cooperative Programming Environment) is a newly completed system devised at Cornell University under the direction of Richard Conway [Ref. 9]. It is intended to offer an alternate system to the CPS environment. COPE is an integrated environment and is capable of immediate execution, a form of reverse execution, and similar features.

The COPE system does much more than ensure program correctness. If incorrect input is received, the system attempts to guess the user's intent and generates syntactically correct code based on that guess. It does not use a template system, nor does it require strict character by character program entry. Because the system is designed to guess a user's intention, only a few tokens which correspond in some way to a program construct need be entered. The system will then supply correct code which corresponds most closely to the fragmented input received. Whether or not this approach goes too far is debatable in its own right and is beyond the scope of this discussion.

COPE utilizes a tree structure to represent programs. Thus, tree movement cursor commands are the rule. However, unlike CPS, program keywords may be directly edited. Any program element is subject to modification in this system. COPE does have a drawback regarding its commenting conventions. Comments are allowed only at the beginning of a block. The implementors felt that good code will be self documenting and that line by line comments would be superfluous.

COPE represents one extreme view of a syntax directed programming environment. It not only assists a user in creating correct code, but will actually try to generate such code for him. Conway stated that users tended to try to "goad" the system into creating desired code sequences rather than attempting to completely specify their own code with the system's assistance. Thus, the design philosophy behind COPE represents the extreme end of the spectrum when compared to systems such as Z and SED.

H. CONCLUSIONS

Very divergent design philosophies are evident in the systems examined. The earliest systems such as Emily required a user to go to extremes when entering a program, but ensured that programs were always syntactically correct. Later systems seem to have focused more on user convenience while trying to provide an atmosphere stimulating the production of correct programs. Total program correctness is not always ensured. The COPE system takes the view that ease of entry and syntactic correctness are equally important goals. Thus, correct programs can be created from very scant user input. The systems share very little common ground in major design decisions. One conclusion then is that the production of syntax directed editors is still a very experimental subject. The chart in Figure 4 depicts the variances among the different systems.

There are a few general properties of syntax directed editors which can be inferred. The editor should be convenient. It should assist the user as much as possible in creating correct and readable programs. It should not be significantly more difficult to use than a conventional text editor. There must be a balance between the facilities provided and their ease of use. The editor should allow an abbreviated means of text entry for the common elements of the supported language. Pretty-printing and holophrasting should be available when desired. The SDE should be able to provide these features common in conventional text editors.

System	User View	Program Entry	Syntax Enforced	Long Range Checks	Immediate Execution	Holo-phrasting	Comment Convention
Emily	Tree	Selection	Yes	Yes	No	Yes	Restricted
MENTOR	Tree	Sequential	No	No	No	Yes	Restricted
Interlisp	Tree	Sequential	Yes	Yes	Interpreted	Yes	Free
CPS	Tree	Both	Yes	Yes	Yes	Yes	Restricted
Z	Text	Sequential	No	No	No	Yes	Free
SED	Tree	Sequential	Yes	No	No	Yes	
COPE	Tree	Fragments	Yes	Yes	Yes	Yes	Restricted

Fig 4. Facilities Comparison

The implementation of these features will require engineering tradeoffs. A baseline syntax directed editor need not be as ambitious as COPE, however to be a true SDE, it should not be a "95% program editor" either. The tradeoffs necessary in a baseline implementation of these features will be more thoroughly discussed in the following section.

III. MAJOR DESIGN DECISIONS

A paramount concern in the design of the system is that it must be able to reside in the small memory common in most microcomputers. This concern must also be weighed against time constraints. An interactive system which provides poor response times will not be tolerated, particularly when the host machine is a dedicated resource.

In most cases, time and space are contradictory resources; one can usually be traded for the other. The design of this system must achieve a balance between the two. The test machine for the editor's implementation has only 58,000 bytes of memory available for transient programs and it uses a relatively slow 2MHz system clock. Because of these system limitations, time and space constraints will impact on every major design decision. They will be implicitly included in the statement of every other design principle.

There is a wealth of information available in the literature regarding properties that contribute to the design of a good system. Two sources have influenced the design effort for this project. W. J. Hansen presented his "User Engineering Principles" [Ref. 2] used in the design of Emily. These are listed in Figure 5. Studies have been conducted at Carnegie-Mellon University (CMU) in an effort to determine a set of properties which constitute a "graceful" interactive system interface [Ref. 10]. The results of their studies are presented in Figure 6. The basic maxim which can be extracted from these sources is that ease of use is a primary concern. The interface for this system assumes a user with little or no knowledge of

Know the user

- Education, Experience, and Patience

Minimize memorization

- Selection, not entry
- Names, not numbers
- Ensure predictable system behavior

Optimize Operations

- Rapid execution of common operations
- Display inertia
- Recognize command parameters

Engineer for errors

- Provide good error messages
- Engineer out the common errors

Fig. 5 Hansen's Design Principles

Flexible parsing -- Correct small, simple mistakes in the interface language

Robust communications -- Avoid verbosity but let the user know when the system understands

Identification from description -- The system can identify internal objects easily

Focus tracking -- The system tracks the user's focus of attention

Explanation facility -- The system can explain what it can do

Personalization -- The system can adjust to the user's desires

Fig. 6 CMU Interface Principles

programming systems. As a result, the interface for the editor will be menu driven as much as possible. Menu selections will be designated by a single letter (possibly accompanied by the control or shift key) which should be mnemonically related to the desired system action. Hansen's "use selection, not entry" principle will be stressed.

Another major principle can be inferred from both sources, the principle of regularity. To ensure that a system is predictable it must be regular in nature. A system responce should be regular across the different possible states of the system. An implication of regularity is that the system should adopt an "all or nothing" approach in the facilities that it provides. A particular design decision that rests on the regularity principle concerns the correctness issue. An approach similar to that of SED should be avoided. Parsing the input "as well as possible" is not a good design decision because the user will not know when he can and can not depend on editor decisions. Thus the time and space spent on partial parsing is wasted in many situations. Either the CPS (all) or Z (nothing) approach should be taken. On the assumption that the editor will be pressed to fit into a small space, program correctness and, thus, parsing will be left for the compiler.

Another important design principle is that the system should do as much as it can for the user. This principle interacts with both of those presented above. Ease of use implies that the user should be freed from mundane, rudimentary tasks which the system can perform. He should be given prompts whenever possible and should have simple things done for him. The regularity principle implies that, while the system should do all that it can, it should not presume too much. If the system is designed to make too many guesses as to the user's intent, it could often guess incorrectly. This would constitute a source of annoyance and would waste the time and space needed to generate the guesses. Thus, the system should try to do all that it can, but it should only attempt those tasks which it can do correctly in most instances.

When considering those things that can always be done correctly, attention becomes focused on those things that are integral to the target language. Such facilities as balancing parenthesis and quotation marks should be included. Further, one knows that the keywords and major constructs of the language must be used in every program. Thus, a natural tendency is to implement some form of the template method of operations. Since the keywords and static constructs of the language are always the same, a template can be used to present their structure in every instance. Further, the use of templates will provide prompts for the user and will allow him to enter many characters with a single keystroke. *The use of templates will correspond to another design principle* as well.

The user should not be required to provide the same information more than one time. Once the system has been given a correct piece of information, the user should be able to 'call up' that same information if it is needed again. Template usage satisfies this principle. The system knows the correct format and construction of every statement and the user is able to employ this knowledge directly. Another facility which will support this principle is a macro substitution feature. Once a user has provided some information to the system he should be able to store it for later reuse. If the system were to save everything automatically, a great deal of space would be wasted. However, if the user knows that he requires repetitive use of a code sequence, he will be able to save it. Only the user will be able to predict his future requirements. A good macro substitution facility can help to "engineer out errors" by allowing correct code sequences to be used many times. Further, only a small amount of memory will be required by this facility.

The system should also be amenable to user desires. It should not dictate user actions. The same freedoms that are available in the conventional text editors should be included in the syntax directed editors as well. This principle has many implications. A full screen editor is a necessity. The teletype approach of MENTOR should be strictly avoided. The editor must provide convenient and powerful text entry commands. The derivation approach of Emily should be discarded. Since arbitrary decisions reflecting the preferences of the designer will not be allowed, the restrictive commenting conventions of CPS and COPE are not appropriate. Because the basic SDE will not attempt parsing or immediate execution, forcing comments to be placed in any specific style is not justifiable, regardless of time and space constraints.

The above discussion presents the primary design principles for this effort. They are a blend of those presented by other system designers and the time and space restrictions inherent to this project. Figure 7 contains a condensed and integrated listing of the major principles.

Adherence to these principles has provided many of the major design decisions. Composing the time and space constraints with these decisions will yield new system traits. Because space is a limited resource and user program correctness is not a goal, internal representation of the user program as an abstract syntax tree would be wasteful. This implies that text oriented commands will be used to move through the programs. The regularity principle and the textual movement format imply that objects created by templates should be subject to editing operations. This situation is possible in a system that does not offer immediate execution or ensure correctness. Allowing templates

Time and space constraints influence every decision

Ease of learning and using the system is important

The system must be regular in the execution of its facilities

The system should include only those facilities that can be done well, not situation dependent features

The user should not be required to restate information

The system should be amenable to the desires of any user

Fig. 7 Major Design Goals

to be edited will also eliminate the need for the cumbersome modification methods of Emily and CPS. A drawback is that a user could 'fool' the editor. Specifically, if the editor knows that a statement is a particular type of template and the user changes the statement, then the editor will not be able to identify the new type of the statement. An integral parser would solve this problem but would also consume space.

Because any construct will be subject to editing and the user should not be bound to editor decisions, an alternative to the template mode of program creation should be available. Further, the editor may not be able to recognize the type of edited objects. Thus a free input mode is a necessity. Such a mode will allow the user to circumvent any editor decision and will enhance ease of program modification. A drawback to this mode of operation is that the editor will not be able to assist the user in any way. It should be noted that such a facility would not be practical in template systems which attempted to ensure program correctness.

Time and space constraints affect one other major design decision. Should the editor be table driven so that many languages can be supported or should the editor be customized to recognize only one language? Clearly, if a table driven model is used, effort will be required to load and interpret the rules of the target language. Again, relying on the assumption that space is a paramount concern, the knowledge of the supported language should be built into the editor. If a good modular design is used, there will be a possibility of supporting many languages by altering only the language specific modules of the editor. Parnas' information hiding and encapsulation principles can be used in this effort [Ref. 11]. The design of the editor should include a separation of functions. This will ensure an efficient use of available space while not totally negating the possibility of multiple language support.

Given these decisions, some language must be selected for the test implementation. The language should be commonly available for microcomputers. Also, the regularity principle implies that the full language should be supported, not a carefully selected subset as is the case with many SDE implementations (CPS and COPE as examples). While support for the full language is desirable, any implementation of a language which includes many additional features should be avoided. The time and space constraints already on the system are restrictive. Supporting a superset of a language could only add to the burden.

Several languages were examined in light of these considerations. The C programming language [Ref. 12] was selected because it adheres to the constraints in almost every implementation. After the detailed design phase of the project began, it became obvious that a better choice could have been made. Although the language is small and a superset was not involved, C is so ill defined that its irregularity hampered the SDE implementation. There does not seem to be a standard grammatical description of the language. Other faults were discovered. The next section of this thesis contains a brief accounting of some of the language dependent problems that were encountered.

Adherence to the design goals and observance of system constraints allowed many design decisions to be made at an early point in time. A concise statement of these decisions is provided in Figure 8.

A full screen editing model will be used

The system will not be table driven

System commands will be single key, menu driven selections

Prompts will be provided as often as possible

Internal storage will be based on a text format

Textual cursor movement commands will be used

There will be no assurance of program correctness

There will be no parsing mechanism

The template mode of program entry will be used and there will be an option
to override this feature

Template structures may be edited

There will be no restrictions on comments

A macro substitution facility will be available

The system will generate punctuation and low level comments automatically

The C programming language will be supported by built in knowledge

Fig. 8 Early Design Decisions

IV. NOTES ON THE C PROGRAMMING LANGUAGE

C [Ref. 12] is a programming language developed at Bell Laboratories by Brian Kernighan and Dennis Ritchie. The language is rapidly growing in popularity and many systems have been implemented in C, UNIX being a prime example. One reason that C has been used for major applications is its power, power in the sense that assembly language is powerful. There is nothing that could be done in assembly language that could not also be done in C. In a major sense, C is simply a portable assembly language which includes some of the convenience features commonly found in modern high level languages. However, because C is so closely linked to assembly language, it contains many of the pitfalls awaiting programmers at that level.

C contains many linguistic traps. Feature interaction and the poorly defined syntax can cause many errors to go unnoticed. These traits caused several problems in the implementation of the syntax directed editor. In order to examine some of the unusual design properties of the language, C will be analyzed in terms of good principles of program language design. Bruce J. MacLennan has defined sixteen such principles [Ref. 13] which are presented in Figure 9. Some of these principles conflict with each other and MacLennan has admitted that no language could satisfy each of them because of these interaction conflicts. However, one would be hard pressed to find a single language which violated more of the principles than does C. In fact, Kernighan and Ritchie intentionally violated some principles in an effort to add assembly level power to the language.

Abstraction -- Avoid requiring something to be stated more than once, factor out recurring patterns

Automation -- Automate mechanical, tedious, error prone activities

Defense in Depth -- If an error penetrates one line of defense, it should be caught by another

Information Hiding -- Ensure that a user has all the information needed to use a module and no more and the implementor has the necessary information to implement a module and no more

Labeling -- Do not require the user to know absolute positions of objects

Localized Cost -- A user should only pay for what he uses

Manifest Interface -- All interfaces should be apparent

Orthogonality -- Independent functions should be controlled by independent mechanisms

Portability -- Avoid features that are dependent on a particular machine

Preservation of Information -- The language should allow the user to present information he knows and the compiler may need

Regularity -- Regular rules, without exception, are easier to learn, use, describe, and implement

Security -- No program that violates the definition of the language or its intended structure should escape detection

Simplicity -- A language should be as simple as possible

Structure -- The static structure of a program should correspond in a simple way to the dynamic structure of computations

Syntactic Consistency -- Things which look similar should be similar, things which look different should be different

Zero, One, Infinity -- The only reasonable numbers are zero, one and infinity

Fig. 9 Maclennan's Principles of Language Design

First, consider the C grammatical specification contained in Appendix A to Reference 12. These syntax charts are preceded by "This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language" [Ref. 12, p.214]. This sentence is a model of understatement. The syntax is presented in a precise notation similar to BNF. However, the precision of this notation is illusionary. Often a single rule having no more than five components will be accompanied by two pages of text which explain the various exceptions to the BNF description as presented. In such cases, the conceptual clarity offered by the notation is lost. The primary example of this situation concerns the rules for variable and type declarations. There are only two basic types in the language. However, seventeen BNF rules and over sixteen pages of text (in summary form) are devoted to explanations of various declarations involving the two types.

As an example, consider the legal and illegal declarations contained in Figures 11 and 12 respectively. Note that, once the preceding type has been added, the objects from both figures are easily derivable from the C declaration rule contained in Figure 10. The C method of specification for legal declarations violates the regularity, security, and simplicity principles and is a gross violation of the syntactic consistency principle. This ill defined nature of legal declaration forms was a major obstacle in the construction of a template for declarations. In this case, abiding by the principle of only doing those things which could always be done correctly, a very free form template was provided. The user receives the very terse prompt "<type><identifier list>" and is free to enter any structure deemed necessary. Even this very basic template is not sufficiently vague to avoid user inconvenience in all cases. If

```
declarator =

        identifier | declarator() |

        (declarator) | *declarator |

        declarator[constant-expression {optional}]
```

Fig. 10 Grammar Rules for Declarators

	F is declared to be a function returning
int F()	an integer
char *F()	a pointer to a character
union **F()	a pointer to a pointer to a union
int (*F()) []	a pointer to an array of integers
struct *(*F())()	a pointer to a function returning a pointer to a structure

Fig. 11 Legal C Declarations

	G is declared to be a function returning
char (G())[]	an array of characters
int (*G()[])()	an array of pointers to functions returning integers
char (*G())()[]	a pointer to a function returning an array of characters
struct (*G())()	a pointer to a function returning a structure

Fig. 12 Illegal C Declarations

a structure object whose members are initialized is declared, this template can be cumbersome. An example of the procedures necessary to effect this type of declaration is provided in the next chapter.

Another major concept in C involves the notion of "lvalues". An "object" is "a manipulatable region of storage" [Ref. 12, p. 183] and an lvalue is "an expression referring to an object" [Ref. 12, p.183]. The concept of lvalues is very basic to the language. Only lvalues may appear on the left side of an assignment statement. Because of the way in which C uses pointers and implicit type coercions, it is often not obvious when something can be considered to be an lvalue. The basic example of an lvalue is an identifier name. However, other lvalues can be easily constructed. As an example, 'a' is a character constant as is 'abcd'. Then *('a' + 'abcd') is a legal lvalue and "*('a' + 'abcd') = 2;" is a legal statement. In this case, 'a' is converted to its integer representation. The characters 'abcd' are also converted to a single integer although the method of conversion varies from machine to machine. The "+" adds two integers. Finally, the "*" tells the compiler to refer to the contents of the following address. Thus, the addition of two characters implicitly coerced into integers is explicitly coerced into an address and becomes a legal lvalue. Such constructions violate many of Maclennan's principles including the security, portability, regularity, defense in depth, simplicity, and the structure principles. Further, the irregularity of such a basic concept as which constructs may appear on the left of the assignment operator causes many problems for the SDE, particularly one which does not include a parser.

C is very amenable to the coercion of variables. There is a strict (although somewhat irregular) hierarchy defining the coercion rules. In the event that automatic coercion is insufficient, a specific operator, the cast, is provided. This allows a user to freely violate the security and defence in depth principles. However, the cast does provide a warning of coercion and is thus supportive of the structure principle.

Another major flaw in the language is that no run time diagnostics are provided. The compilers will be very content to generate code which effects an out of bounds array assignment. These assignments can, in fact, store values into locations where executable code should reside. Thus, mysterious errors can occur and there will be no clue as to their origin. Such actions violate many principles, including the automation principle. Initially, an attempt was made to enable the editor to generate in line code which would provide notice of out of bounds assignments. However, this task proved to be extremely difficult without a knowledge of the dynamic state of the program in execution. Notice that in Figure 10, the bounds description in an array declaration may be omitted in some cases. Generally, omission is allowed in the first index position or when the array is explicitly initialized. An array declared as a formal parameter which has no bounds declaration inherits the bounds of the actual parameter. The editor would have no way of determining these bounds from the static program. Thus, the feature could not be applied in all cases and was omitted.

Arrays themselves can cause much confusion. When an array is declared, the cardinality of its possible members must be used to define the array bounds. That is, "char arrayname[9]" declares an array of 9 characters. However, a reference to arrayname[9] is an out of bounds reference. Array subscripts start at 0 while size declarations start at 1. Further, the major use of arrays in C is to hold character strings. Thus, if an array was designed to hold the string "cat", one would assume that "char arrayname[3]" would be a proper declaration. This is not the case. The compilers recognize strings and automatically insert a 0 into the position following the last character in the string. Thus, the declaration must account for one additional place and "char arrayname[4]" would be the proper declaration in this example. Referring to arrayname[3] would yield the 0 byte and arrayname[2] the letter 't'. This situation violates many principles and again frustrated SDE attempts to provide some protection for the user.

The operators which designate assignment and equality tests represent another poor choice made by the C implementors. The '=' is the assignment operator while '==' is used to designate equality tests. A natural act for a user constructing a program is to type what he thinks. Thus, it would be very easy to enter "while(a = b)..." which would cause an infinite loop as long as the value of b was nonzero. The operator choice in this case violates the orthogonality and syntactic consistency principles. The editor was able to provide some assistance in this case. Because of the frequency with which the error occurs, a specific check for assignments when a conditional expression was expected was included.

In general, many of the operator conventions implemented in C represent violations of the structure, regularity, and syntactic consistency principles. One example concerns the cascading of operators within a single statement. When code such as "a=b=c=0" is used, the cascading feature is convenient, both for the user and the compiler. However, the cascading feature also applies to relational operators. A legal statement is "a<b<c". This piece of code would

be very convenient if its meaning corresponded to its appearance. This code will first test a and b. If a is less than b, the first part of the statement will evaluate to 1, 0 otherwise. Then, the value of c will be tested against 1 or 0. A further example concerns the overloading of operators, as exemplified by the '&' operator. Dependent on context, a single '&' can mean "the address of" or "bitwise logical and". Also, '&&' means "logical and" and "x &= 1" means "x = x & 1", a bitwise logical and operation.

The comma operator represents the worst C operator overloading convention. A single comma may have three meanings. First, it separates parameters in function calls. Secondly, it separates different expressions within a single statement. Finally, it is used as a value separator in an initialization list. The combination of these meanings can cause problems. The statements "call(x,y)" and "call((x,y))" are totally different. The first is a function call with two arguments, the second call has only one parameter. The use of the comma in the second statement supposes that x has some side effect which is complete in itself. Once evaluated, the value of x is discarded and the value of y is used as the actual parameter. The comma convention has implications for compilers which attempt to build syntax trees immediately, eliminating parenthesis as they are encountered. Such compilers would evaluate both of the above statements as function calls with two arguments [Ref. 14].

Finally, the '++' and '--' operators deserve some mention. These are the increment and decrement operators. They are based on a machine instruction found in the machine on which C was originally implemented (a violation of the portability principle). These operators can be applied only to lvalues and if used as "++x" have the meaning "x = x + 1". Logically then, one would assume that "++(++x)" would have meaning "x = (x + 1) + 1". However, "++(++x)" is an illegal construct because (++x) is not an lvalue. Further, these operators have different meanings depending on their relative positions. The meaning of "++x" is increment x and then use its value. Conversely, "x++" means use the value of x and then increment x. Thus, the statements "y = ++x" and "y = x++" have two very different meanings although they look similar. Further confusion is possible if these operators are applied to pointers. Then the increment value is no longer 1, but the size in terms of bytes of storage of whatever is referred to by the pointer. Clearly, the increment and decrement operators violate the structure and syntactic consistency principles.

C does not support information hiding or encapsulation very well. The language is composed only of functions. A function can return, at most, a single value. Thus, global variables are often required to make efficient use of function calls. A related point of confusion concerns functions. All actual parameters are passed by value, except arrays, structures, and unions. These may only be passed by reference. This is but one of many exceptions to generally stated rules in C.

A good example of exceptions to the basic rules in C concerns the definition of what constitutes a legal identifier name. The rule may be described as follows. The first character must be a letter. Capitalized letters are different from lower case letters. Identifier names may be any length, however only the first eight characters are significant, unless the identifier has an external storage class in which case only the first seven positions may be significant and lower case may be considered the same as upper case, depending on the machine being used. This rule is not extraordinary at all and, it would seem, that the makeup of an identifier name is a basic element which should be portable across many machines without exception.

There are many other examples which show the irregularity of the C language. Some scoping rules are based on the type of the declared object. There are poorly designed major constructs such as the switch statement. The language offers implicit and explicit aliasing of variables. However, continued discussion of these and similar issues is not the primary intent of this thesis.

This chapter has been included in an effort to provide some insight into the difficulties encountered in designing and implementing a syntax directed editor for the C language. At the start of the project, it was hoped that some degree of protection for the user might be built into the editor. However, the possible interaction of many of C's irregular features made this goal extremely difficult. A few protective measures were included in the editor, but not as many as was originally intended.

The purpose of this chapter is not to damage the reputation of a language which enjoys widespread use. As stated earlier, C should be considered as a high level assembly language. As such, C is a very good tool for its intended purpose. However, a more regular language would aid the implementation of a SDE. A serious effort was made to produce a strict grammatical definition of the language. The results of that effort are contained in Appendix A.

V. C-SMART USER'S MANUAL

C-Smart is intended to behave exactly as its name implies. The editor is smart, but it is not brilliant. It is able to do many things automatically and it tries to assist the user in several ways. However, the editor can be circumvented if the user so desires. The most salient feature of C-Smart is its template system. There is a template for every construct in the C programming language, including the preprocessor commands. The punctuation required by each construct can be automatically provided by the editor. The template system has five major subsystems. These are the preprocessor, declaration, function definition, statement, and comment template components.

The template system is very closely linked to two other major components of the editor, the input monitor component and the macro substitution component. Both of these components interact in very basic ways with each of the template subsystems. In order to understand the template operation, it is necessary to first comprehend the input monitor and macro substitution components. Thus, these two components will be discussed first.

A. INPUT MONITOR

The input monitor controls all user input to the editor. It serves as a line editor within the overall screen editor. The input monitor operates on the bottom three rows of the screen in the user input area. This area is separated from the rest of the screen by a dashed line, the separation line, which has its own function. The line provides a mechanism to display the nesting level of the current line of the program. C relies on the convention of using the "{" and "}" characters to mark the beginning and the ending of blocks. When a new block is entered, the editor will display the "{" character on screen at the appropriate indentation level. Because the entire program can not be displayed on screen at one time, it is easy to lose track of nesting levels. Thus, whenever the editor displays the opening "{" on screen, the matching "}" is displayed on the separation line at a corresponding indentation level. At block exit, the "}" is moved from the separation line to the screen and a comment is appended to the "}" indicating the type of block being exited (e.g. "} /* end for */" would be displayed). This action is done automatically. Because there are only eighty characters on the separation line, the nesting level indications are limited to a depth of sixteen blocks.

The input monitor recognizes several user commands. These are distinguished from text entry by use of the control key (represented graphically as '↑'). The control key and the command key should be depressed simultaneously to enter a command. Figure 13 lists the control keys recognized by the input monitor.

When the input monitor recognizes a ↑a command it will attempt to add everything from the current cursor position to the end of the user input line as a new macro definition. If two ↑a's are recognized on the same line, the input monitor will attempt to add everything between the two marks as a new definition. If three or more ↑a's appear on the same line, the value of the closest mark already defined will be adjusted. Only one macro definition may be made for each user input line. The ↑i command will cause the editor to ignore any marks previously set on that line. If additional ↑a commands are not used, no new definition will be made on that line. The ↑u command instructs the input monitor to use one of the pre–defined macro definitions. There are two ways to identify the desired substitution. The user should either enter enough letters of the definition to disambiguate it from any other definition or enter a

↑c	exit to the operating system
↑m	return
↑a	add a macro definition
↑i	ignore macro definition request
↑u	use a macro definition
↑d	delete the character to the left of the cursor
↑e	delete to end of line
↑x	x–out the entire line
↑s	move the cursor to the start of line
↑n	move the cursor to the end of line
↑l	move the cursor one character left
↑r	move the cursor one character right

Fig. 13 Input Monitor Commands

number corresponding to the position on screen of the desired substitution. If the first method is used, a blank must follow the disambiguating letters. When a blank is not desired in the input, the number designation method may be used. The substitutions are numbered 1 to 17 from left to right, top to bottom as they appear on screen. When the ↑u key is pressed, the editor will insert a '@' character at the current cursor position to mark the location where the substitution will be inserted. When the return key is pressed, the user input line will be moved to the screen and the '@' character will be replaced by the indicated substitution. Examples of using the macro substitution feature are provided in later discussion.

The cursor movement commands operate exactly as indicated in Figure 13 and require no further explanation. If the editor recognizes the ↑c command, control will be immediately passed to the operating system. Files in use will not be saved or closed. When the ↑m or, more conveniently, the return key is pressed, the input monitor will return control to the editor. The return key signals that the user has completed his actions. It is often used to signal that no further selection is desired from the current menu.

The ↑d command deletes the character immediately to the left of the cursor. The ↑e command deletes all user input from the current cursor position right-ward to the end of line. The ↑x command will delete the entire user input line regardless of cursor position. A fresh input line will be supplied after the deletion.

There are several keys which the input monitor does not recognize. A beep is sounded whenever one of these keys is pressed. The unrecognized keys are those which have an ASCII value of less than 32 and have not been designated as command keys. Several examples are the delete, backspace, and tab keys.

A key function of the input monitor is to control line length. The C language allows logical lines to occupy more than one physical line when the '\' line continuation character appears at the end of the continued physical line. The editor will recognize lines which are too long and will insert continuation characters as needed.

The editor also controls indentation. Blank spaces will be inserted in front of the user input when the screen is refreshed. Thus, the user need never concern himself with the indentation required on any line. The editor displays the '*' character in the user input area to show the user how much physical space, minus leading blanks, is available on the current physical line. If the user enters input beyond the '*' mark, the editor will insert the continuation character, display the first part of the logical line, and present a fresh input line in the user input area so that the logical line may be continued. The process of extending logical lines across several physical lines may be continued indefinitely. However, unreadable programs could easily result.

B. MACRO SUBSTITUTION COMPONENT

The substitution component is intended to allow the user to enter correct input only once and use it repeatedly. Information can be called from the substitution facility any number of times and at almost any point in the editing process. There are two areas of storage reserved for the substitution component. These are initialized to support declarations and statement entry options. The content of either area may be modified or used by means of the ↑a and ↑u keys respectively, as discussed in the previous section.

The declaration support storage values will be visible at any time that the current line of the program could contain a declaration. The predefined values are the standard types and storage classes available in C. When these values are visible, they are displayed on the top five lines of the screen, just above the text area. Only the first few characters (11) of any visible item will be displayed, regardless of the actual length of that item. This restriction is due to the limited screen area available to display the view definitions.

The presentation of the statement support values is similar. The predefined values are functions available in the C standard library. These definitions are visible whenever the editor expects a statement to be entered on the current line.

The user is free to define any construct to be stored for future use. The limitations are that there may only be 34 such definitions extant at any time and that any single definition may not exceed fifty characters in length. If an attempt is made to store longer strings. the editor will notify the user. In such a case, the user should break up his storage requirement into two or more definitions.

Only one construct per user input line may be saved in a macro storage area. Once a construct has been successfully designated for storage, the user will be asked which one of the currently visible definitions should be replaced. The user should respond by providing a number, 1 thru 17, corresponding to the position, on screen, of the view item to replace. Subsequently, the visible definitions area of the screen will be updated to reflect the currently stored values.

The substitution component is a convenient means of referencing frequently used constructs. It may also be used to temporarily hold a value during editing. As an example, if the user wishes to surround a statement with a while template, the substitution facility can hold that construct and then reinsert it at the appropriate point within the while template. This process provides an alternative to the free input mode of program modification (to be discussed later).

C. TEMPLATE COMPONENTS

The template system has five major subsystems. Each of these contain pre-defined templates appropriate to some part of the C language. The largest of the subsystems is the statement template subsystem.

The statement subsystem holds templates for each statement type in the C language. A menu of these templates will be presented whenever a statement could be entered as the current input line. To select a particular template, the user holds the shift key and presses the key corresponding to the menu designation of his choice. The selected template will then be displayed on screen and the user will begin to fill in the parts of the template.

The only exception to this procedure is the expression template because expressions are the lowest level and most frequently used construct in the language. Thus, they should be very easy to select. The editor does hold a special expression template which may be selected. However, this can be cumbersome given the frequency of expression usage. To ease expression entry, the editor adopts the convention that anytime a lowercase letter is struck when the statement menu is available, the user wishes to enter an expression. The editor automatically selects the expression template and accepts the user input as an expression. Again, the editor adopts this convention only when the statement menu is available to the user.

The comment template is intended to support comment blocks. The opening and closing comment characters will be automatically inserted on every physical line when the comment template is in use. This template is also able to accomplish word wrapping. If user input exceeds the line space available, the editor will break that line at the last complete word, add the close comment characters to the line, display the line on screen, insert the leftover portion of user input into a fresh input line, and allow the user to continue his input. These actions are performed automatically.

Comments may be inserted without using the comment template. Any template will accept a comment as an integral part. Word wrapping will also be performed on comments entered with other constructs. Further, if the start comment characters are entered on any input line, and no close characters are also present, the editor will assume that the line ends in a comment and add the close comment characters.

The function definition template does not operate in a menu driven manner: there is only one template available for selection. When the user selects the function definition template, he receives the "<type> <function name> (<parameter list>)" prompt. The type component is optional and assumed to be 'integer' if not supplied. The shortest user input under this template would be of the form "fname (". On this input, the editor would add the closing ")" indicating an empty parameter list, display the input, display the opening block "{" character, and move to the local declaration template to await user input. A more common class of input would be of the form "fname(a, b, c" where a, b, and c are the formal parameters. On input of this type, the editor would again add the closing ")" and display the user input. The next action would be to display 'a' in the user input area and ask for a type designation. Once provided, 'a' and its type would be displayed. Then 'b' would be displayed in the user input area and the "<type> or @" prompt would appear. If the user entered a type, then 'b' and its type would be displayed on the next physical screen line. If '@' were entered, then 'b' would be added to the line holding the declaration of 'a', indicating that 'a' and 'b' were of the same type. This process would continue until the formal parameter list was exhausted. Then, the local declaration template would be provided.

The declaration template also operates without a menu because there is only one template available. This template has the very basic form "<type> <identifier list>". It is very difficult to supply a template for every possible legal C declaration. Thus, the user is free to enter anything he desires under the declaration template. The only restriction is that the storage class and type should be entered first and the identifiers and any initializers next. This template can be cumbersome in some cases. As an example, to declare a struct with initializers, the user should enter 'struct', the struct name and the opening "{" when given the <type> prompt. When the <identifier> prompt is presented, the user should enter the type and name of the first struct member. Then, on subsequent <type> and <identifier list> prompts, the type and name of struct members should be entered. When the last struct member is specified, the user must explicitly add the closing "}" and any identifiers declared to be struct's of the type described.

The global and local declaration templates are basically the same, having only two small differences. Auto and register macro definitions are not available when declaring global variables. Also, the local declaration template will not accept initializers for variables which do not have the static storage class. These restrictions are in keeping with C language requirements.

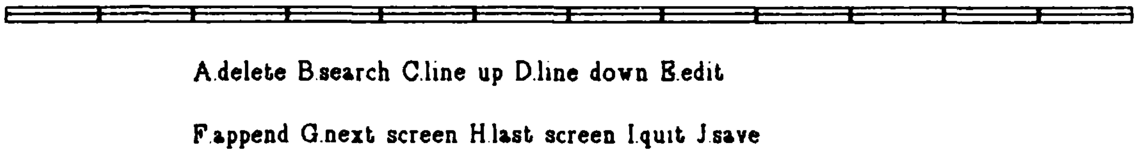
The preprocessor template subsystem operates in a manner very similar to the statement subsystem. There is a template available for every preprocessor command available in standard C and any one may be selected from a menu. As with the statement templates, the editor will add ending punctuation. One exception to this rule occurs when the "#asm" template has been selected. The editor does not have any knowledge of assembly languages. Thus, no assistance is available under this template. The user is entirely responsible for ensuring that the requirements of his assembler are met.

D. MENUS

As discussed, most template selections are menu driven, the only exceptions occurring when only one choice is possible. Whenever a menu is available, it is displayed on the top two lines of the user input area. Except when entering an expression under the statement menu, the user should select an option by typing the capitol letter corresponding to the desired menu object. In general, once an item has been selected, its template will appear on screen and the user will begin to receive prompts to complete the various subparts of the template. If an illegal selection is made, the user will be so advised and requested to enter his selection again. If the user enters no choice (return key only), the menu will end and the preceding menu will reappear on screen.

The top level menu does not work directly with the template system. It offers the options as depicted in Figure 14. The cursor up, cursor down, next screen, and last (i.e. previous) screen commands operate as their titles imply. Cursor movement commands will not present new screens however. The next and last screen commands must be used for this purpose. The cursor ('>') is displayed in the far left margin of the line that the editor recognizes as being the current line.

The append option provides the means of text entry. When selected, text will be inserted after the current line. Thus, the cursor should be positioned appropriately before the append option is selected. If text is to be entered as the first line in the file, the cursor should be positioned at the blank line at the top of the first screen. After the append option has been selected, the user will be asked if he desires the template system or the free input mode. If



A.delete B.search C.line up D.line down E.edit

F.append G.next screen H.last screen I.quit J.save

Fig. 14 Top Level Menu Options

the template system is selected, the editor will determine which type of construct may appear as the current line, set up an appropriate level of indentation, and present a menu for template selection. If free input mode is selected, the template system will be bypassed entirely. The user is free to enter any construct as the editor operates very much like a conventional editor in free input mode. Thus, the user is responsible for all indentation and punctuation decisions. Free input is intended to ensure that program modifications can be accomplished as easily as possible. A caution is that the editor will not recognize the type of any construct entered as free input. Subsequent modifications to these constructs can not be made in the template mode.

The editor contains a very basic search command. Only the buffers currently in memory can be searched. If a global search is required, the user will have to manually cause the buffers to be exchanged and repeat the search in each buffer. The search command does support an 'entity search' option, primarily intended to locate identifiers. Often, the characters that constitute an identifier name may appear as a substring of additional tokens. Conventional editors will return every match for a search object, regardless of whether or not that match is a unit by itself. Entity search will find a matching string only if it is an entity itself. This search mode is possible because the editor knows the syntax of C. When a possibly matching string is found, the characters before and after the matching sequence are inspected. If either character is a legal identifier character, no match is returned. If both characters are legal separators (i.e. +, *, &, etc.) then a match is returned. An entity search is requested by typing the '@' character in front of the target string. This character will not be considered when searching. If the '@' does not precede the target string, a normal search is performed and all matches are returned. Once a target has been found, the line containing that target is moved to the user input area and may be edited. When the user presses the return key, the search is continued. This process repeats until the entire buffer in memory has been searched. If no match is found, no lines will be moved to the user input area.

The last two options available from the top level menu are quit and save. The difference between them is that quit is a faster means of exit because it does not generate a file suitable for input to a compiler. The editor works with a file which contains fixed size pages so that text can easily be moved to and from the disk. Each line in this file carries with it data which allows the editor to determine the type of that line and other pertinent information. Thus, the editor must process this file into a form suitable for a compiler, striping off the header information and deblocking each line.

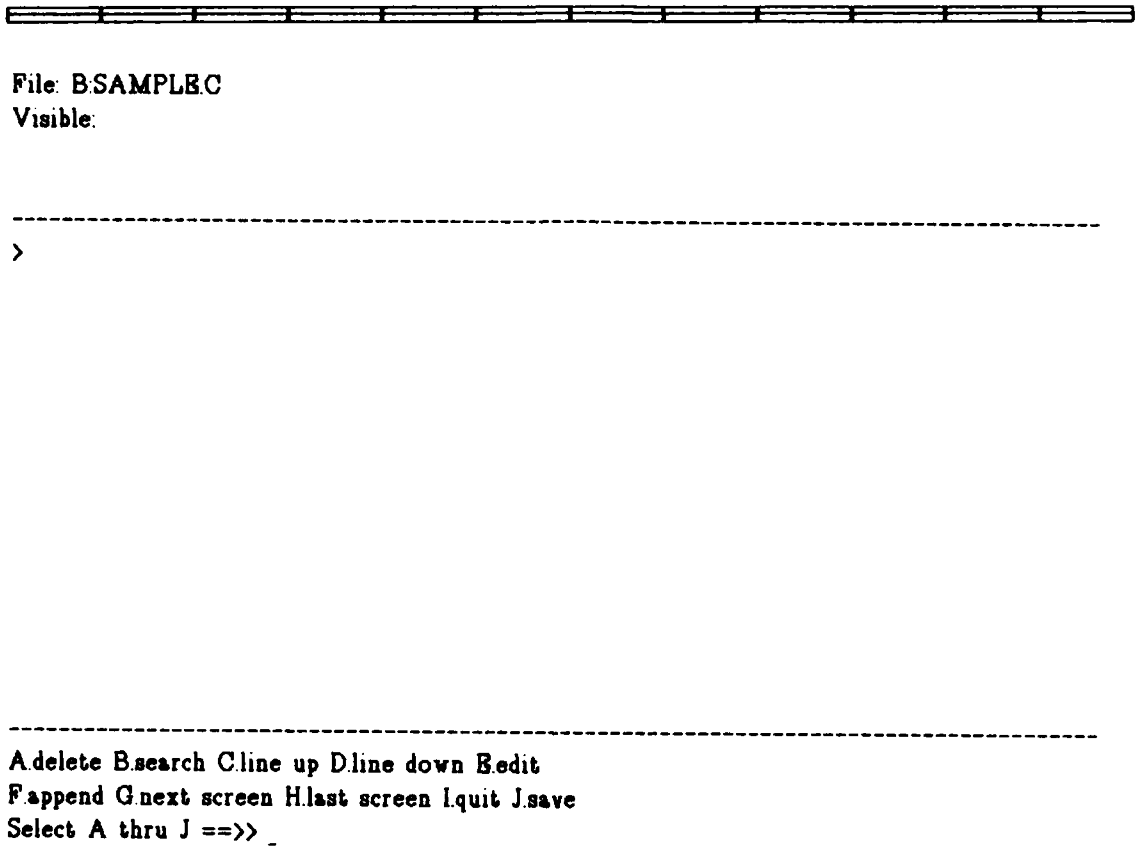
The editor is only capable of working on files which abide by the above format rules. Thus, the editor can only be used on files that it has created. The files which are used by the editor can be recognized by their ".dcs" extension. These files should not be printed because they contain information which is in integer rather than ASCII format.

E. A SAMPLE SESSION

This section contains a sample session with C–Smart. Program constructs from Reference 12 have been selected to be entered as a new program being created by the editor.

The disk containing C–Smart should be placed in the 'A' disk drive. The file to be edited may reside on any drive. This example session will create a file named "sample.c" on the 'B' disk drive. To invoke the editor, select the 'A' drive and enter "CS B:SAMPLE.C" *followed by a carriage return.* Initially, the message "C–Smart, Ver 1.0 NEW FILE" will appear on the screen. Then, the screen will clear and be refreshed to appear as illustrated in Figure 15.

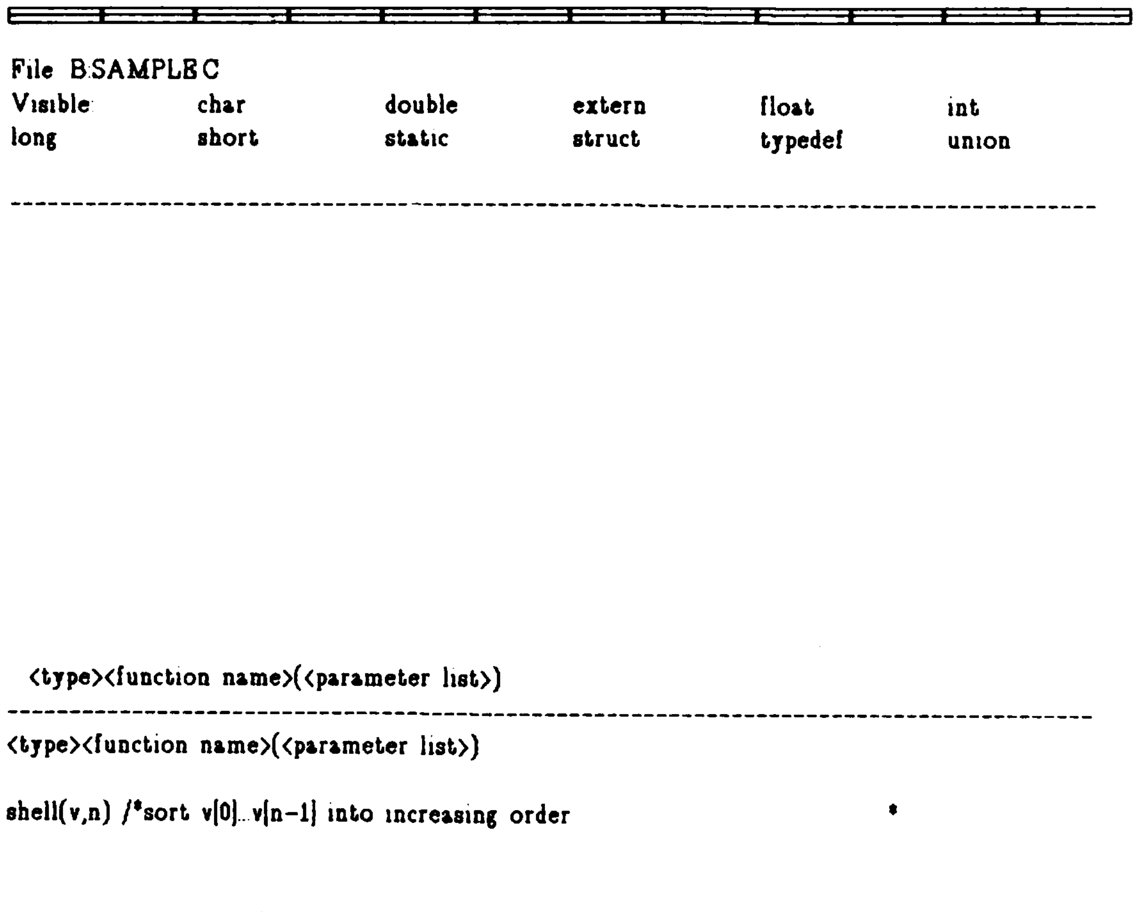
Figure 15 and all other figures depicting screens are slightly off scale. The dashed separation lines are longer in the figures than they actually appear when they are displayed on the computer monitor. The top line of Figure 15 displays the name of the file being edited. The next three lines display the macro substitutions that are currently visible. Notice that Figure 15 has no macro substitutions visible and, thus, none are available. This is because the editor is at its top level and no statement can be entered at this point. The cursor ('>') is positioned at the first line of the file. Sample.c is a new file, so its first line is blank. Below the bottom dashed line, the top level menu is being displayed and the user is being prompted for a selection. In this example, append was chosen to create a new file.



```
File: B:SAMPLE.C
Visible:


--------------------------------------------------------------------------------
>









--------------------------------------------------------------------------------
A.delete B.search C.line up D.line down E.edit
F.append G.next screen H.last screen I.quit J.save
Select A thru J ==>> _
```

Fig. 15 C-Smart Initial Screen

Choosing the append option when creating a new file does not produce the option of free input. Instead, the user is presented with template system options to enter a block comment, preprocessor commands, global declarations, or a function definition. A function definition template was chosen. Then, the function name, parameter list, and a comment were entered. Figure 16 depicts this situation, just prior to pressing the return key. Notice that macro definitions are now visible. These are types and classes that may be used in the definition of a function. Also, the '*' character appears at the far right of the user input line. This is a mark generated by the editor to inform the user of the physical space remaining on the current line. There are also two identical constructs on the screen, one above and one below the bottom dashed line. The top construct is a placeholder. When the user has completed his input, it will appear on screen at the position of the placeholder. The construct

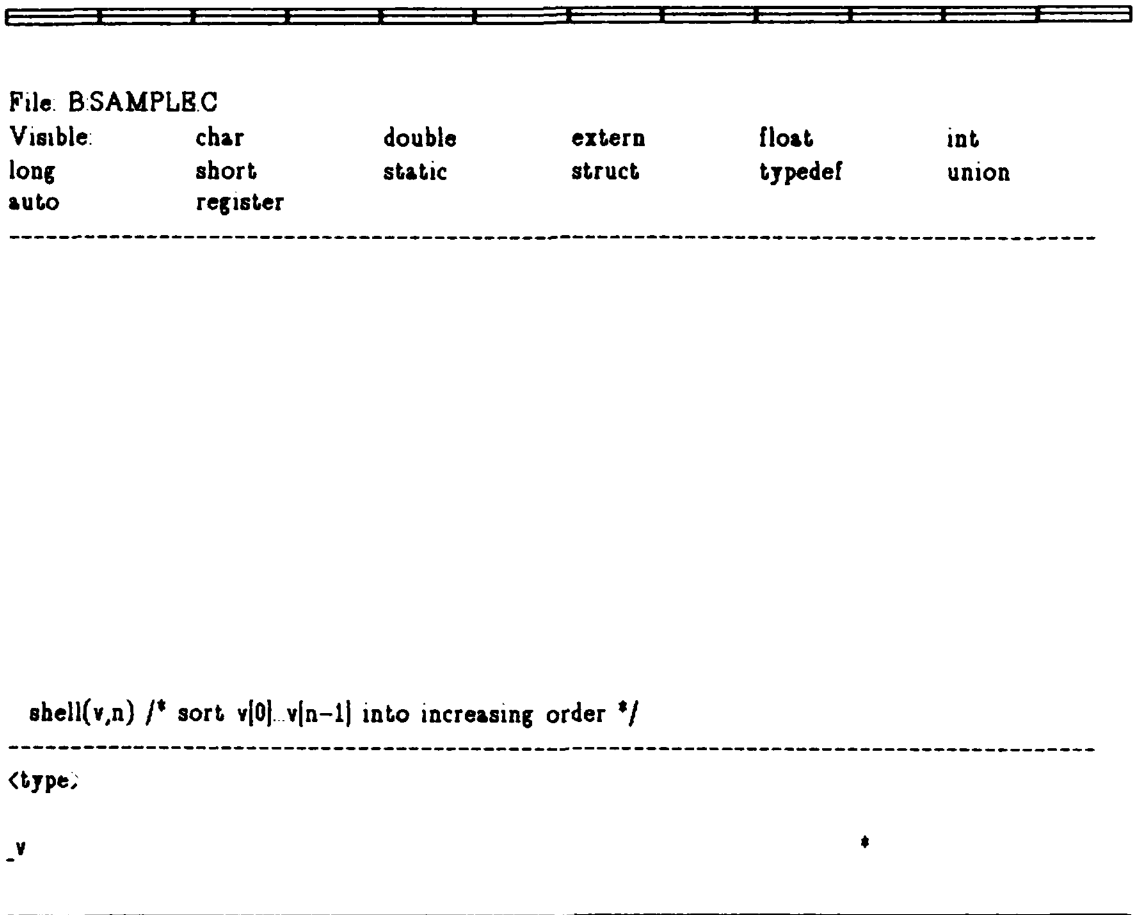


```
File B:SAMPLE.C
Visible     char        double      extern      float       int
long        short       static      struct      typedef     union

-----------------------------------------------------------------------------------











 <type><function name>(<parameter list>)
-----------------------------------------------------------------------------------
<type><function name>(<parameter list>)

shell(v,n) /*sort v[0]...v[n-1] into increasing order                 *
```

Fig. 16 C-Smart Screen 2

below the dashed line is a prompt. This is the syntactic element that the editor expects the user to enter. A function definition is a rare case where the two constructs are identical. In most cases, the prompt is only a part of the placeholder on the screen.

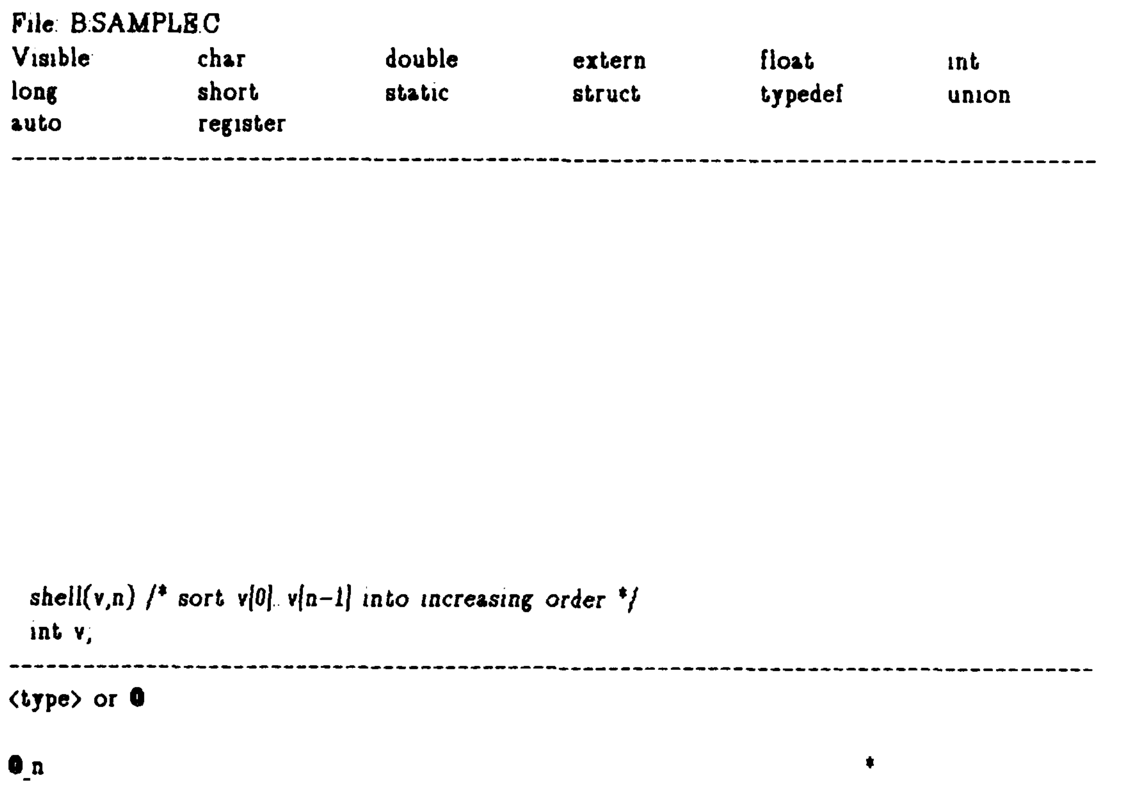
The user strikes the return key and the screen changes as depicted in Figure 17. First, notice that the closing comment characters have been added.



```
File: B:SAMPLE.C
Visible:        char           double         extern         float          int
long            short          static         struct         typedef        union
auto            register
-----------------------------------------------------------------------------------













  shell(v,n) /* sort v[0]...v[n-1] into increasing order */
-----------------------------------------------------------------------------------
<type>

_v                                                                   *
```

Fig. 17 C-Smart Screen 3

Then, the user has been prompted for a <type> (just below the bottom dashed line). Also, the cursor in the user input area is positioned just before the character 'v', the name of the first formal parameter. One final change is in the 'Visible:' area. Two new definitions have been added. This is because "auto" and "register" variables are not allowable in global declarations (as in functions), but they may be used in local declarations.

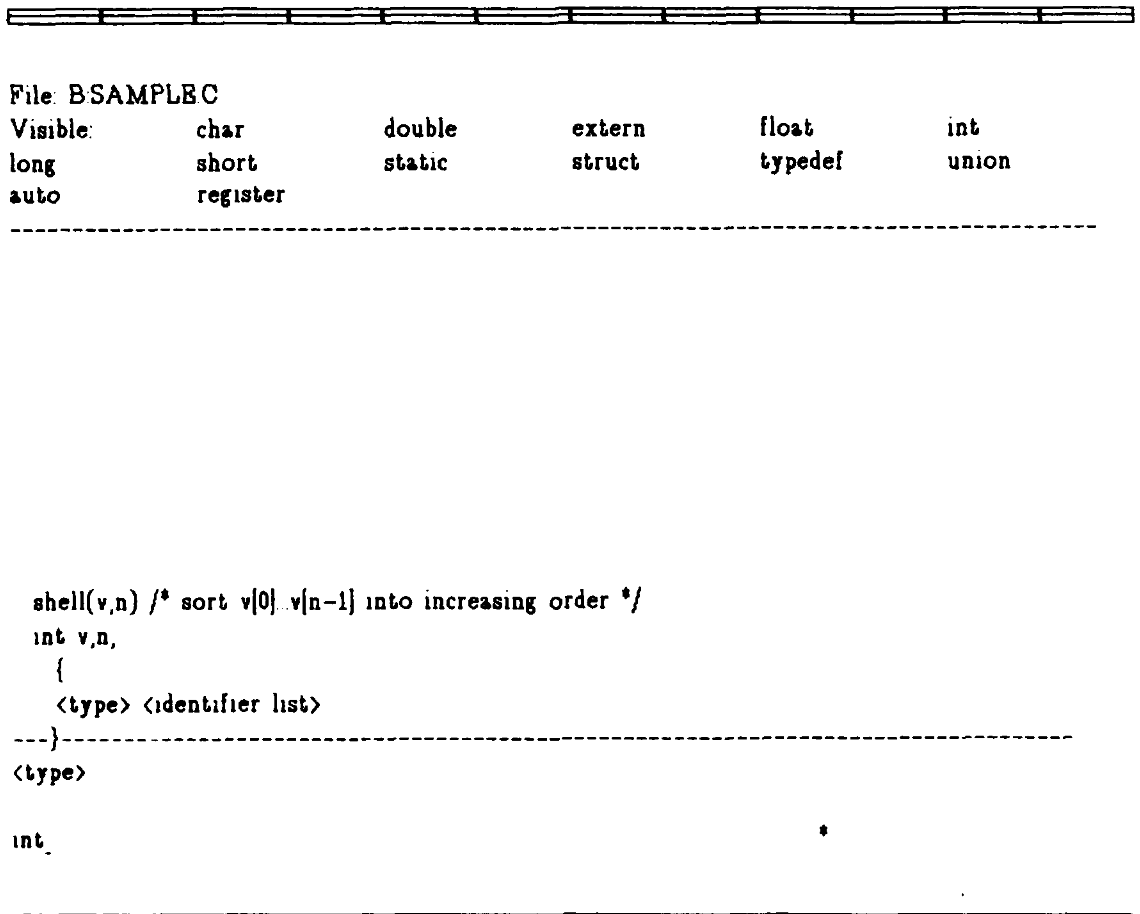


```
File: B:SAMPLE.C
Visible         char            double          extern          float           int
long            short           static          struct          typedef         union
auto            register
-------------------------------------------------------------------------------------











  shell(v,n) /* sort v[0]..v[n-1] into increasing order */
  int v;
-------------------------------------------------------------------------------------
<type> or @

@_n                                                                      *
```

Fig. 18 C-Smart Screen 4

The user enters the type int and strikes the return key, producing the screen as displayed in Figure 18. The declaration of 'v' has been moved to the screen and the user is receiving the "<type> or @" prompt for the formal parameter 'n'. Both 'n' and 'v' are integers, so the user has entered '@'. Pressing the return key produces the screen depicted in Figure 19. Here, the formal parameter list has been exhausted, so the editor has entered the opening block

"{" and activated the local declaration template. Note that a matching close block "}" character is displayed on the bottom separation line.



```
File: B:SAMPLE.C
Visible:        char          double        extern        float         int
long            short         static        struct        typedef       union
auto            register
-----------------------------------------------------------------------------------









  shell(v,n) /* sort v[0]...v[n-1] into increasing order */
  int v,n;
    {
    <type> <identifier list>
---}-------------------------------------------------------------------------------
<type>

int_                                                                        *
```

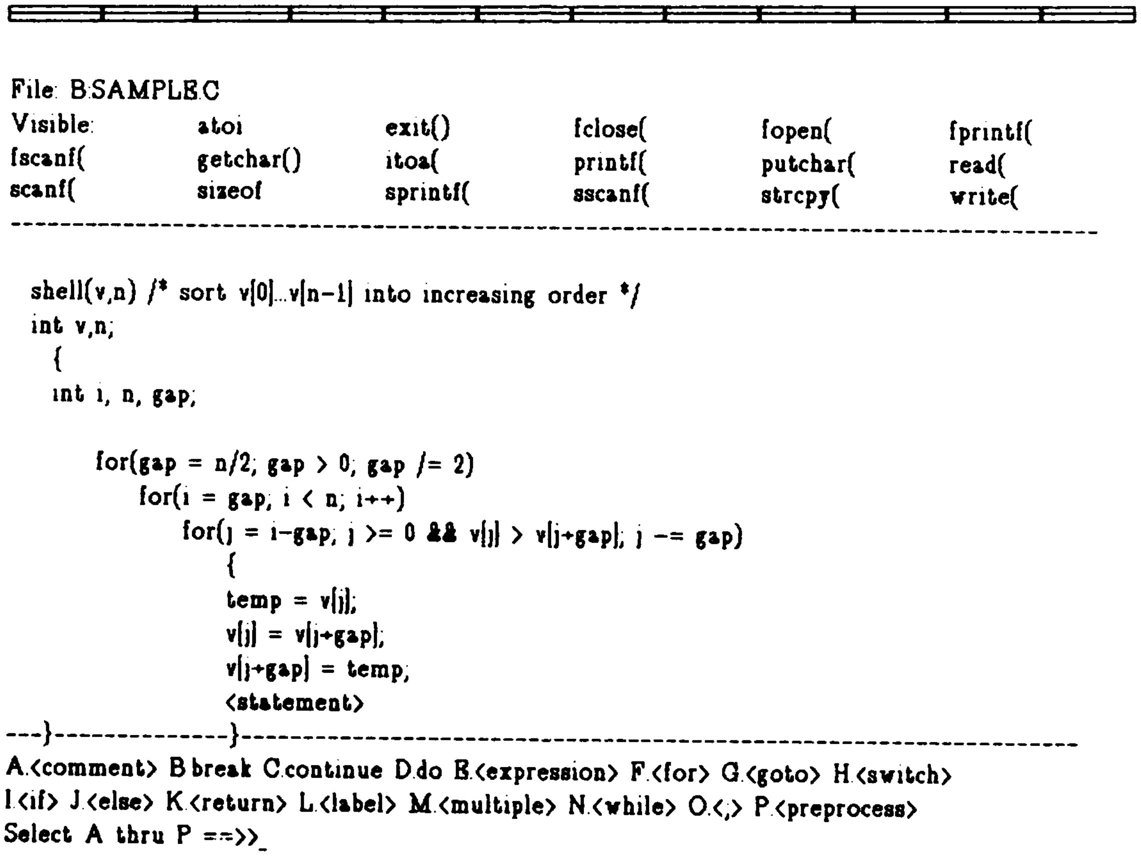
Fig. 19 C-Smart Screen 5

The user continues to enter his program until the state of Figure 20 has been reached. Here, the user is in the middle of entering a for statement. The last remaining incomplete part of the for template is "<expression>". Notice that a "<statement>" is an integral part of the for template. This fact holds true for many statement class templates. Also, notice that a new macro definition is visible. These pre-defined values are available whenever a statement may be entered. As a final comment regarding Figure 20, notice that the end of line marker on the user input line is much closer to the left margin. This is a reflection of the available space remaining on the current (physical) screen line.

```
File: B:SAMPLE.C
Visible:        atoi           exit()         fclose(        fopen(         fprintf(
fscanf(         getchar()      itoa(          printf(        putchar(       read(
scanf(          sizeof         sprintf(       sscanf(        strcpy(        write(
------------------------------------------------------------------------------------




 shell(v,n) /* sort v[0]...v[n-1] into increasing order */
 int v,n;
   {
   int i, n, gap;

       for(gap = n/2; gap > 0; gap /= 2)
           for(i = gap; i < n; i++)
               for(j = i-gap; j >= 0 && v[j] > v[j+gap]; <expression>)
                   <statement>
---}----------------------------------------------------------------------------
<expression>

j -= gap_
```

Fig. 20 C-Smart Screen 6

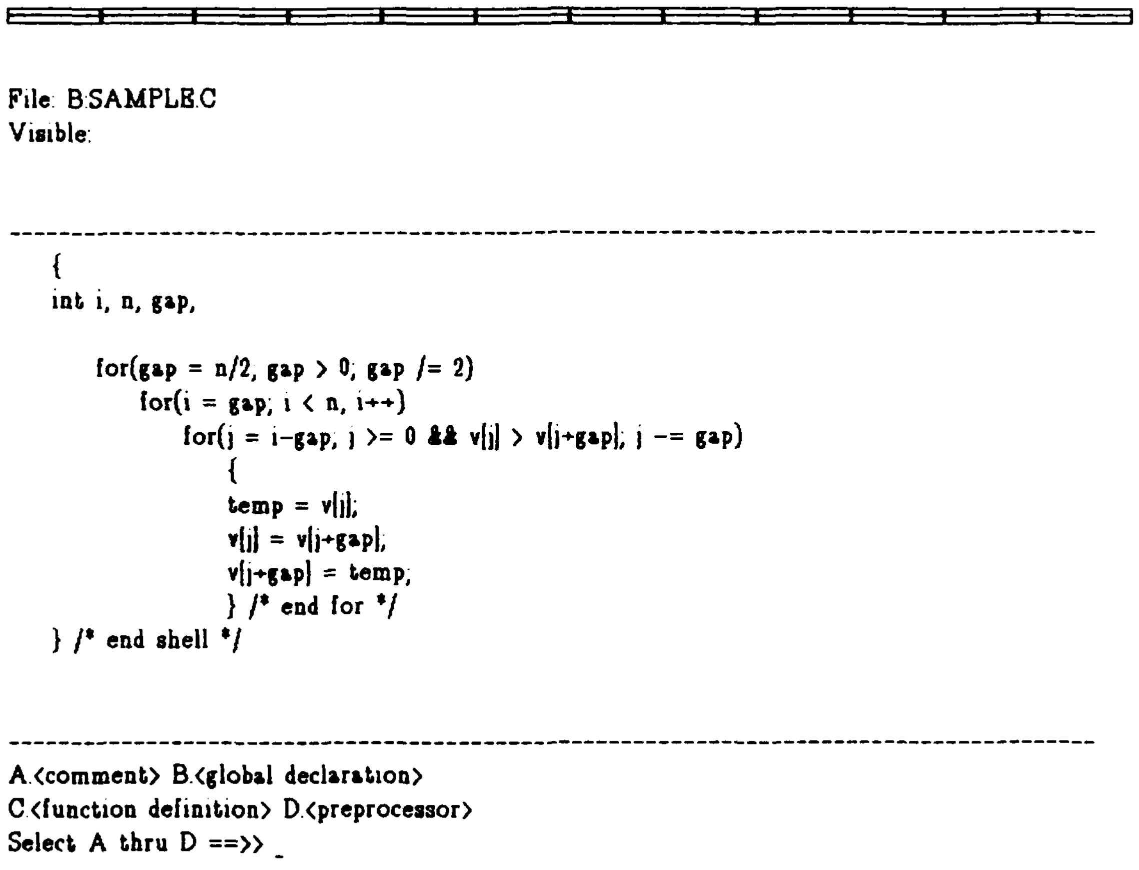
Again, assume that the the user has continued his input up to the point reflected in Figure 21. Here, two "}" characters are present on the bottom separation line and the statement menu is available. The function "shell" is complete and the user is ready to move to the next function definition. The user presses the return key one time to exit the statement menu. Then, the rightmost "}" is moved from the separation line to the screen and the statement placeholder reappears at an indentation equal to that of the "for(gap = 2..." statement. Again, the user strikes the return key and the screen is refreshed to resemble Figure 22. At this point, no macro definitions are visible because the top level menu is available. Also, note that appropriate comments have been added to the closing block characters and that two blank lines have been generated to separate function "shell" from the next program construct.



```
File: B:SAMPLE.C
Visible:        atoi            exit()          fclose(         fopen(          fprintf(
fscanf(         getchar()       itoa(           printf(         putchar(        read(
scanf(          sizeof          sprintf(        sscanf(         strcpy(         write(
--------------------------------------------------------------------------------------

  shell(v,n) /* sort v[0]...v[n-1] into increasing order */
  int v,n;
    {
    int i, n, gap;

        for(gap = n/2; gap > 0; gap /= 2)
            for(i = gap; i < n; i++)
                for(j = i-gap; j >= 0 && v[j] > v[j+gap]; j -= gap)
                    {
                    temp = v[j];
                    v[j] = v[j+gap];
                    v[j+gap] = temp;
                    <statement>
---}-------------}---------------------------------------------------------------
A.<comment> B.break C.continue D.do E.<expression> F.<for> G.<goto> H.<switch>
I.<if> J.<else> K.<return> L.<label> M.<multiple> N.<while> O.<;> P.<preprocess>
Select A thru P ==>>_
```

Fig. 21 C-Smart Screen 7

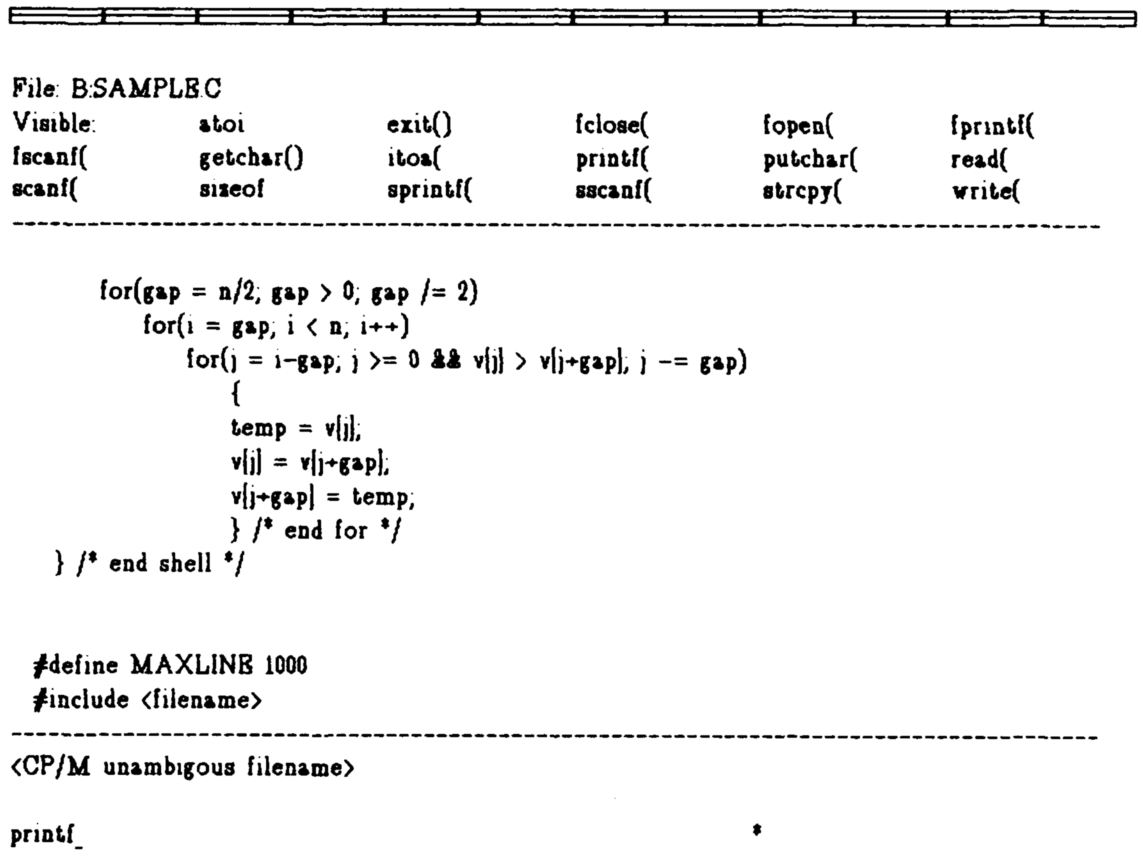
Figure 23 depicts the situation when the user is including the standard library file "printf.c". He is using the preprocessor template and has received the prompt requesting a filename. The user has entered only "printf". The editor assumes default extensions of ".c" and that quotes (as opposed to angle brackets) will be used to specify the location of the file. Thus, the editor generates the included filename as illustrated in Figure 24. In Figure 24, the



```
File: B:SAMPLE.C
Visible:

--------------------------------------------------------------------------------
  {
  int i, n, gap,

      for(gap = n/2, gap > 0; gap /= 2)
          for(i = gap; i < n, i++)
              for(j = i-gap, j >= 0 && v[j] > v[j+gap]; j -= gap)
                  {
                  temp = v[j];
                  v[j] = v[j+gap];
                  v[j+gap] = temp;
                  } /* end for */
  } /* end shell */

--------------------------------------------------------------------------------
A.<comment> B.<global declaration>
C.<function definition> D.<preprocessor>
Select A thru D ==>> _
```

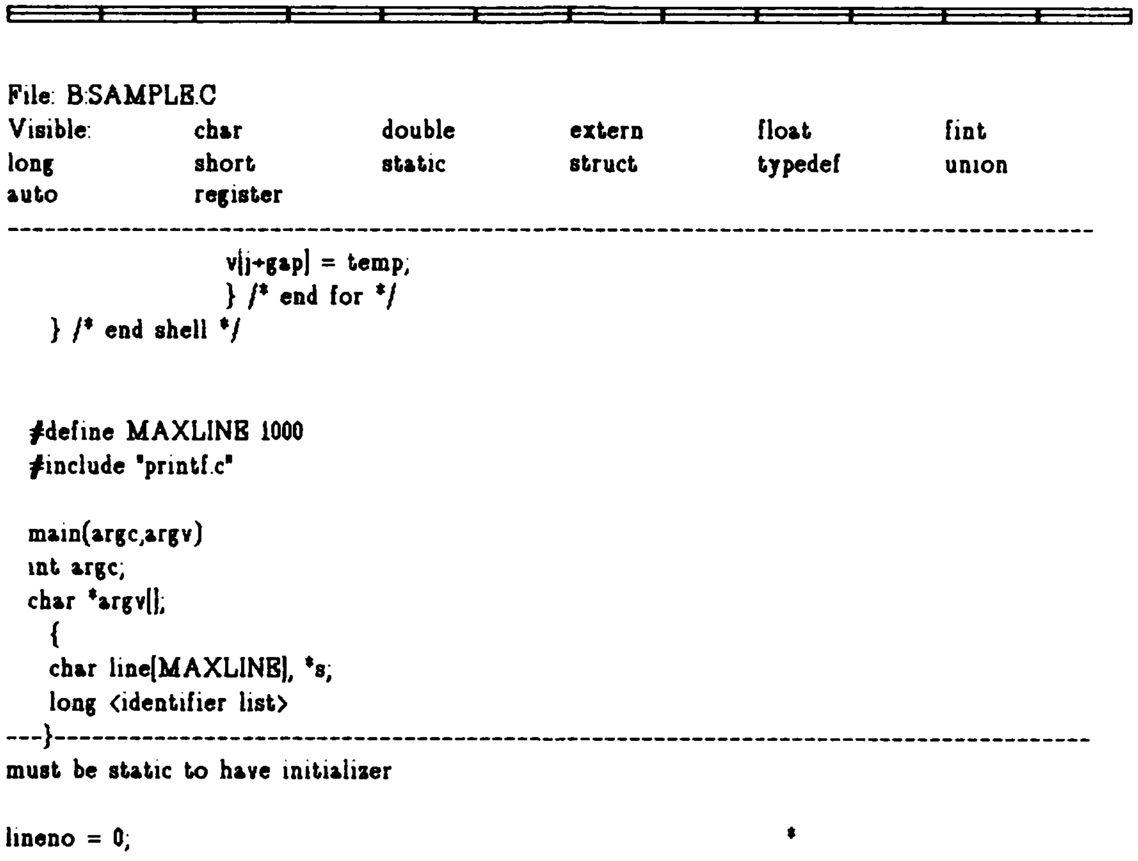
Fig. 22 C–Smart Screen 8

user is entering local declarations to the function "main". He has provided a type of "long" for the current variable "lineno". The user has also tried to initialize that variable. The editor has responded that the variable must be declared as "static" to be initialized and has repositioned the user input on the user input line for editing. At this point, the user should insert the static storage class into his declaration before proceeding.



```
File: B:SAMPLE.C
Visible:        atoi          exit()        fclose(       fopen(        fprintf(
fscanf(         getchar()     itoa(         printf(       putchar(      read(
scanf(          sizeof        sprintf(      sscanf(       strcpy(       write(
--------------------------------------------------------------------------------

        for(gap = n/2; gap > 0; gap /= 2)
            for(i = gap; i < n; i++)
                for(j = i-gap; j >= 0 && v[j] > v[j+gap]; j -= gap)
                    {
                    temp = v[j];
                    v[j] = v[j+gap];
                    v[j+gap] = temp;
                    } /* end for */
    } /* end shell */


  #define MAXLINE 1000
  #include <filename>
--------------------------------------------------------------------------------
<CP/M unambigous filename>

printf_                                                       *
```

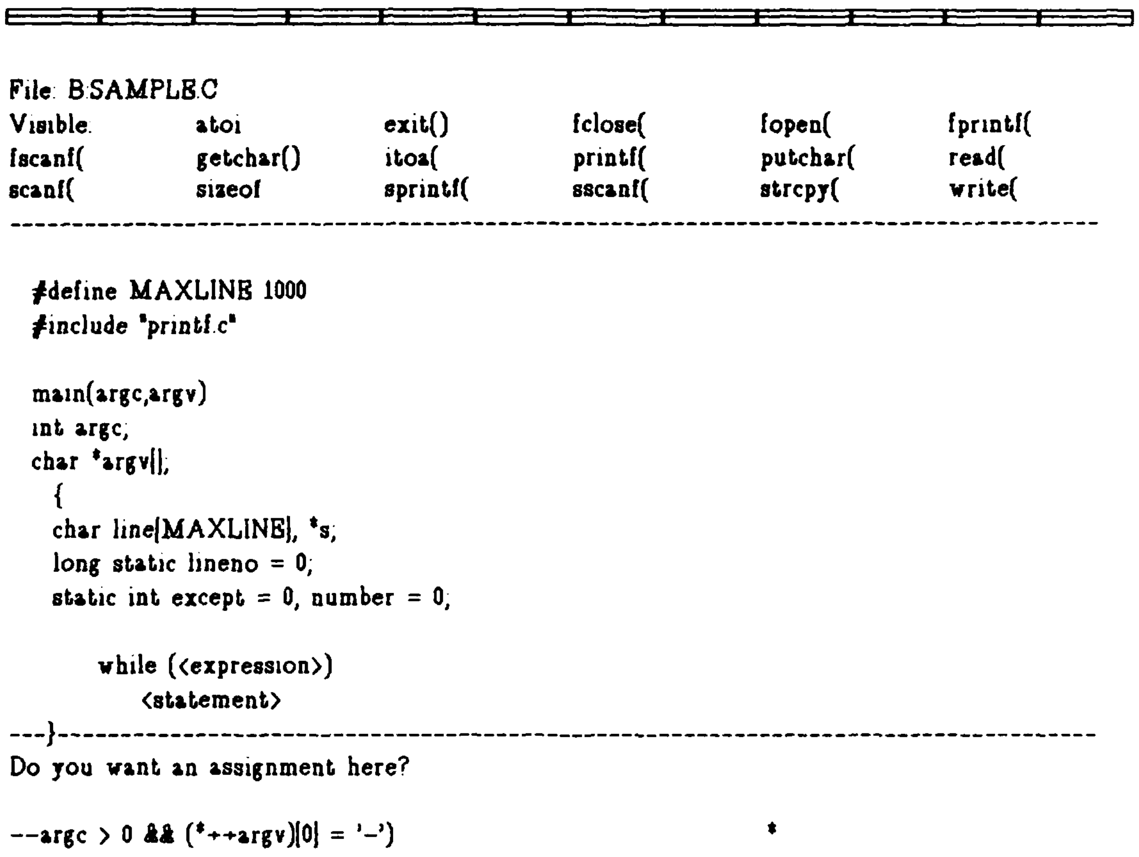
Fig. 23 C–Smart Screen 9



```
File: B:SAMPLE.C
Visible:        char        double      extern      float       fint
long            short       static      struct      typedef     union
auto            register
--------------------------------------------------------------------------------
                    v[j+gap] = temp;
                    } /* end for */
    } /* end shell */


  #define MAXLINE 1000
  #include "printf.c"

  main(argc,argv)
  int argc;
  char *argv[];
    {
    char line[MAXLINE], *s;
    long <identifier list>
---}----------------------------------------------------------------------------
must be static to have initializer

lineno = 0;
```

Fig. 24 C–Smart Screen 10

Figure 25 depicts an interesting property of the editor. The user has entered an expression to be used in a while template. T.e editor knows that the input should be a conditional expression. It examines the user input and determines that an assignment expression has been entered. This is a very

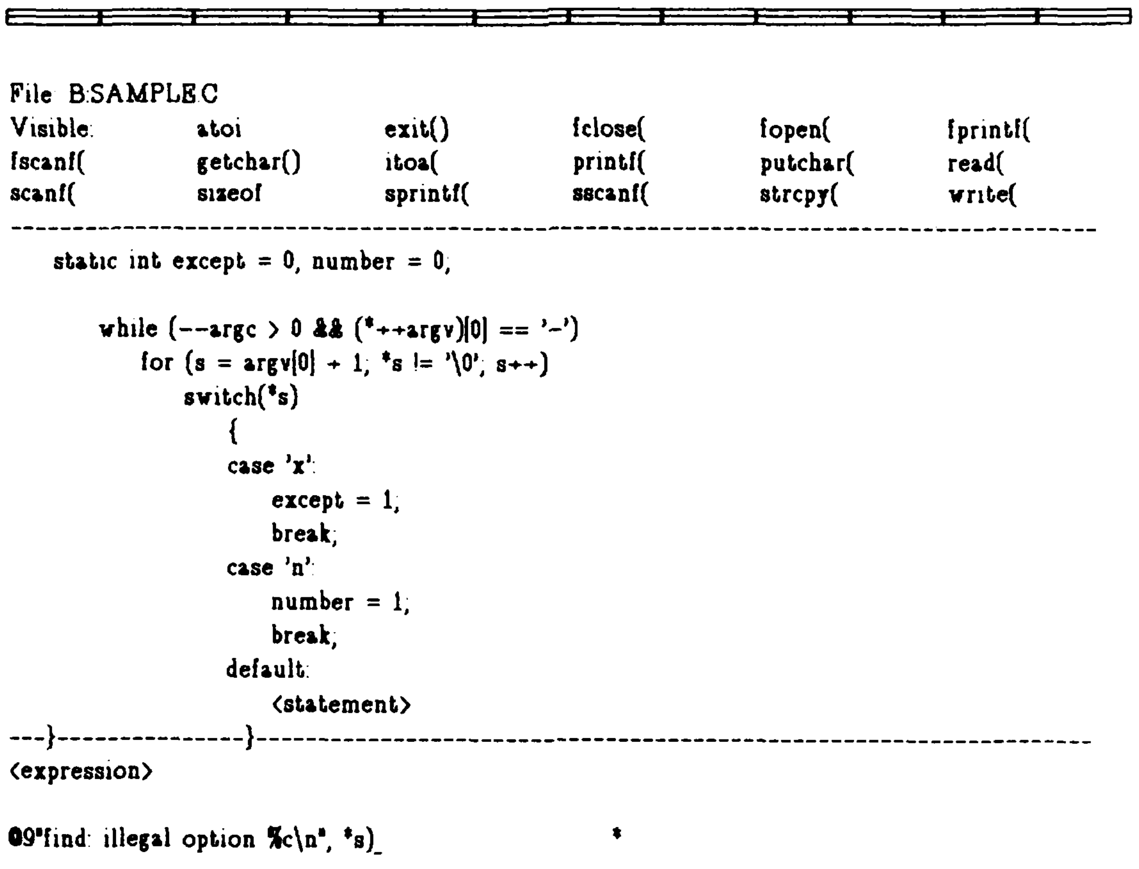


```
File: B:SAMPLE.C
Visible:    atoi        exit()      fclose(     fopen(      fprintf(
fscanf(     getchar()   itoa(       printf(     putchar(    read(
scanf(      sizeof      sprintf(    sscanf(     strcpy(     write(
-------------------------------------------------------------------------

  #define MAXLINE 1000
  #include "printf.c"

  main(argc,argv)
  int argc;
  char *argv[];
    {
    char line[MAXLINE], *s;
    long static lineno = 0;
    static int except = 0, number = 0;

        while (<expression>)
           <statement>
---}---------------------------------------------------------------------
Do you want an assignment here?

--argc > 0 && (*++argv)[0] = '-')
```

Fig. 25 C–Smart Screen 11

frequent error made in C which can cause infinite loops. Recognizing this, the editor repositions the input on the user input line with the prompt "Do you want an assignment here?". The user may edit his input or, by only pressing the return key, leave it unchanged and continue his editing session.

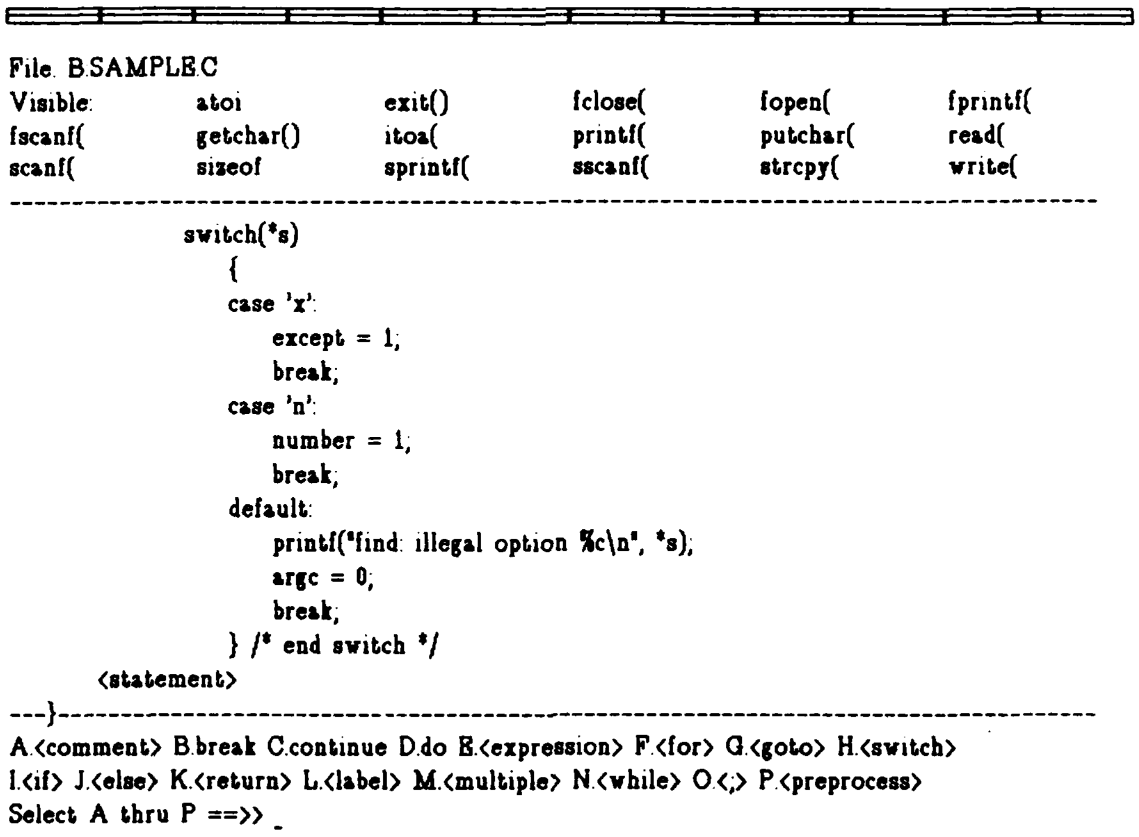
Figure 26 exhibits the macro definition property of the editor. In this case, the user wishes to access the function "printf". Instead of typing the



```
File B:SAMPLE.C
Visible:        atoi            exit()          fclose(         fopen(          fprintf(
fscanf(         getchar()       itoa(           printf(         putchar(        read(
scanf(          sizeof          sprintf(        sscanf(         strcpy(         write(
------------------------------------------------------------------------------------------
   static int except = 0, number = 0;

      while (--argc > 0 && (*++argv)[0] == '-')
         for (s = argv[0] + 1; *s != '\0'; s++)
            switch(*s)
               {
               case 'x':
                  except = 1;
                  break;
               case 'n':
                  number = 1;
                  break;
               default:
                  <statement>
---}--------------}-----------------------------------------------------------------------
<expression>

@9"find: illegal option %c\n", *s)_                    *
```

Fig. 26 C-Smart Screen 12

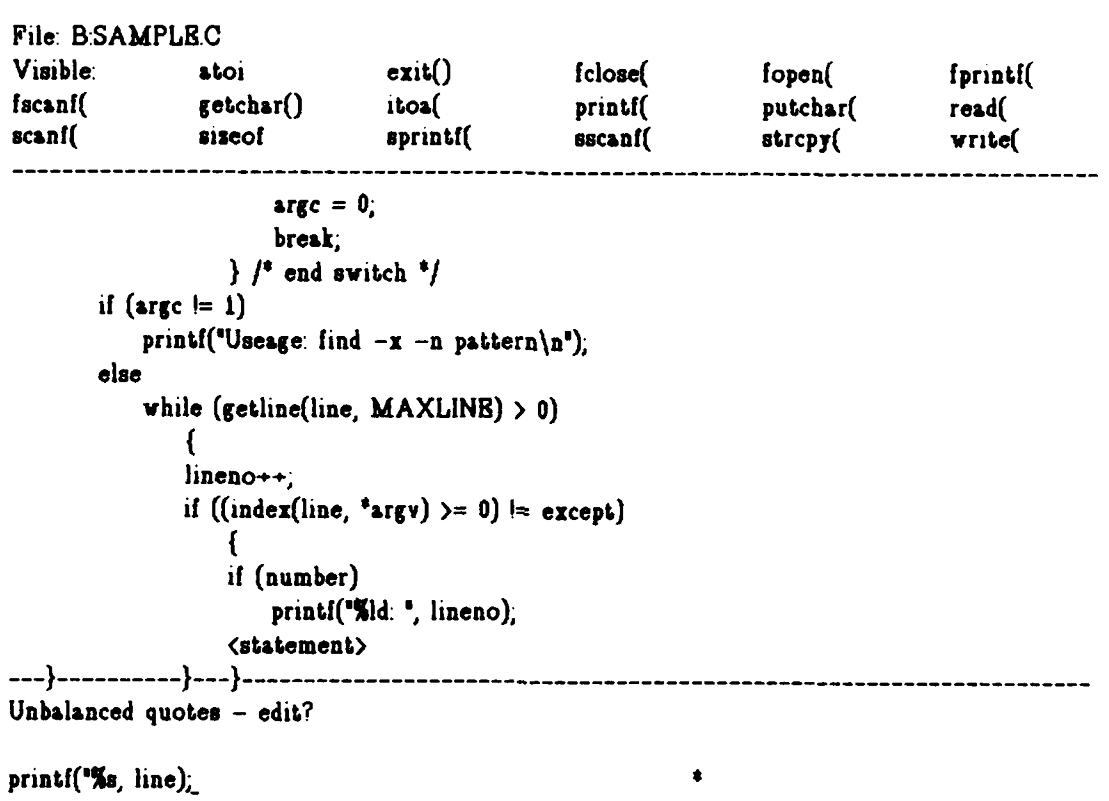
function name, a substitution will be made. The editor generated the '@' character when the user issued the ↑u command. Then, printf was designated by the number 9. When the return key is pressed, "printf(" will be substituted for the characters "@9". Figure 27 depicts the screen after the substitution has been made.



```
File. B:SAMPLE.C
Visible:        atoi            exit()          fclose(         fopen(          fprintf(
fscanf(         getchar()       itoa(           printf(         putchar(        read(
scanf(          sizeof          sprintf(       sscanf(         strcpy(         write(
-------------------------------------------------------------------------------------
                switch(*s)
                    {
                    case 'x':
                        except = 1;
                        break;
                    case 'n':
                        number = 1;
                        break;
                    default:
                        printf("find: illegal option %c\n", *s);
                        argc = 0;
                        break;
                    } /* end switch */
        <statement>
---}---------------------------------------------------------------------------------
A.<comment> B.break C.continue D.do E.<expression> F.<for> G.<goto> H.<switch>
I.<if> J.<else> K.<return> L.<label> M.<multiple> N.<while> O.<;> P.<preprocess>
Select A thru P ==>> _
```

Fig. 27 C–Smart Screen 13

Figure 28 demonstrates a check that the editor can make for the user. The editor can ensure balanced parenthesis, double quotation marks, and single quotation marks. In Figure 28, the user has entered a single quotation mark. The editor has discovered this fact and has given the user an opportunity to

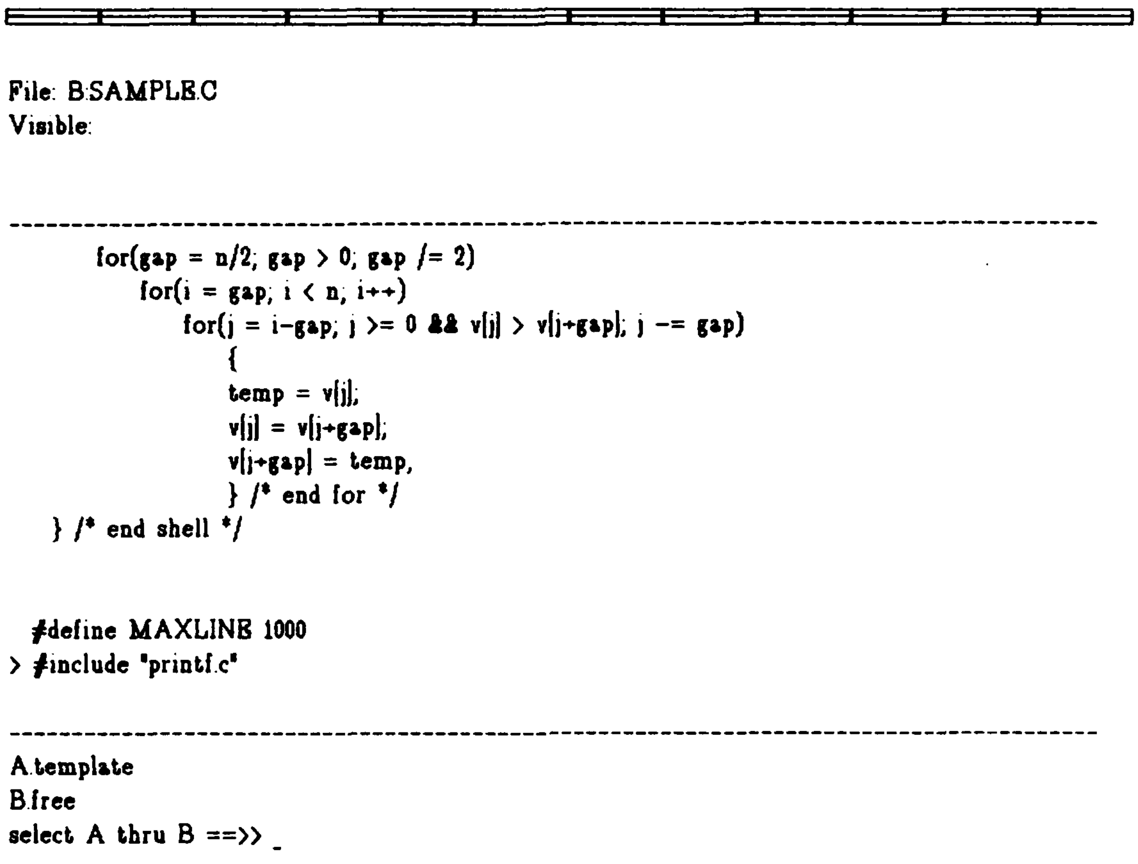


```
File: B:SAMPLE.C
Visible:     atoi         exit()       fclose(      fopen(       fprintf(
fscanf(      getchar()    itoa(        printf(      putchar(     read(
scanf(       sizeof       sprintf(     sscanf(      strcpy(      write(
-------------------------------------------------------------------------
                   argc = 0;
                   break;
                } /* end switch */
      if (argc != 1)
         printf("Useage: find -x -n pattern\n");
      else
         while (getline(line, MAXLINE) > 0)
            {
            lineno++;
            if ((index(line, *argv) >= 0) != except)
               {
               if (number)
                  printf("%ld: ", lineno);
               <statement>
---}---------}---}-------------------------------------------------------
Unbalanced quotes - edit?

printf("%s, line);_                              *
```

Fig. 28 C-Smart Screen 14

correct the situation. He is free to edit the line for corrections. He can also ignore the situation by pressing the return key. This option may be needed if one logical line has been extended over two physical lines and the matching quotation mark is on another physical line.

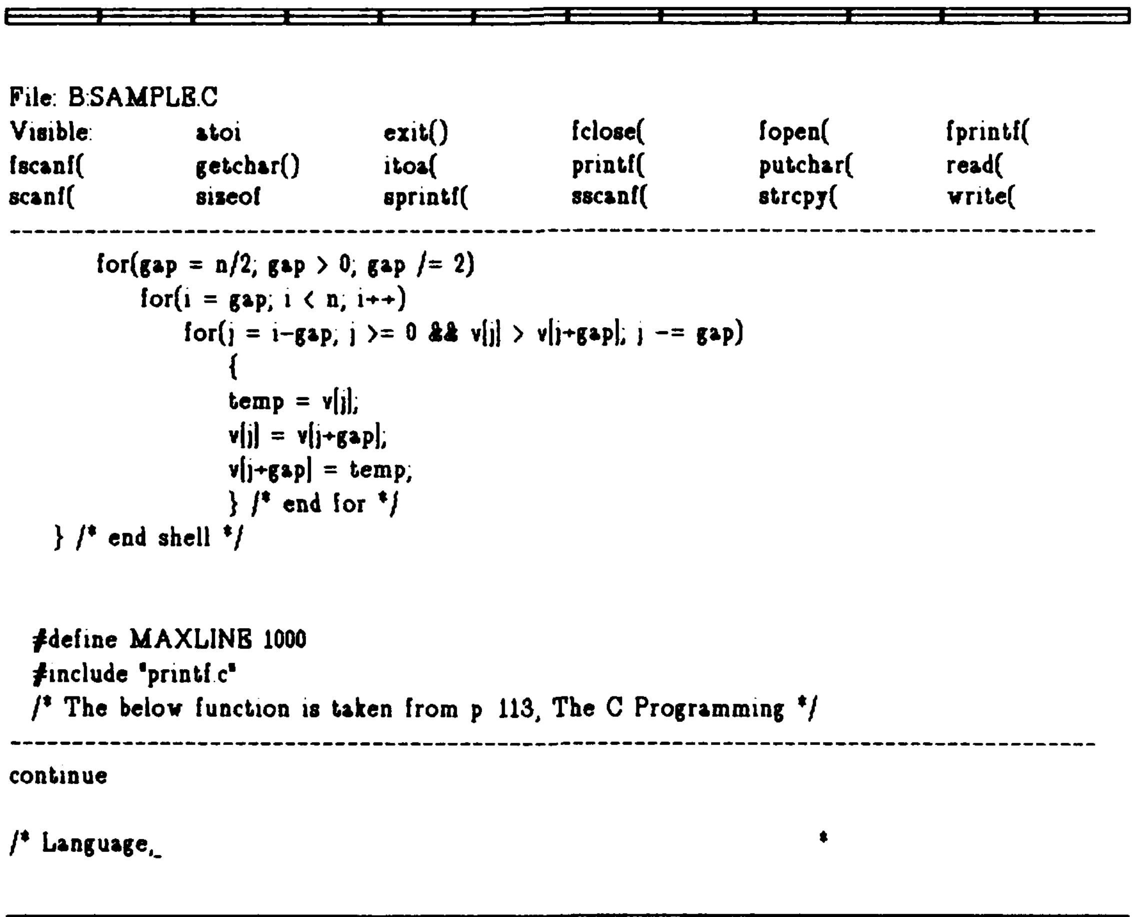
Figures 29 and 30 demonstrate the modification capabilities of the editor. The user has decided to add a block comment before the function "main". To do so, he returns to the top level menu, moves the cursor to the "#include" line and selects the append option. The editor responds by presenting the



```
File: B:SAMPLE.C
Visible:

----------------------------------------------------------------------------------
        for(gap = n/2; gap > 0; gap /= 2)
            for(i = gap; i < n; i++)
                for(j = i-gap; j >= 0 && v[j] > v[j+gap]; j -= gap)
                    {
                    temp = v[j];
                    v[j] = v[j+gap];
                    v[j+gap] = temp,
                    } /* end for */
    } /* end shell */


  #define MAXLINE 1000
> #include "printf.c"

----------------------------------------------------------------------------------
A.template
B.free
select A thru B ==>> _
```

Fig. 29 C-Smart Screen 15

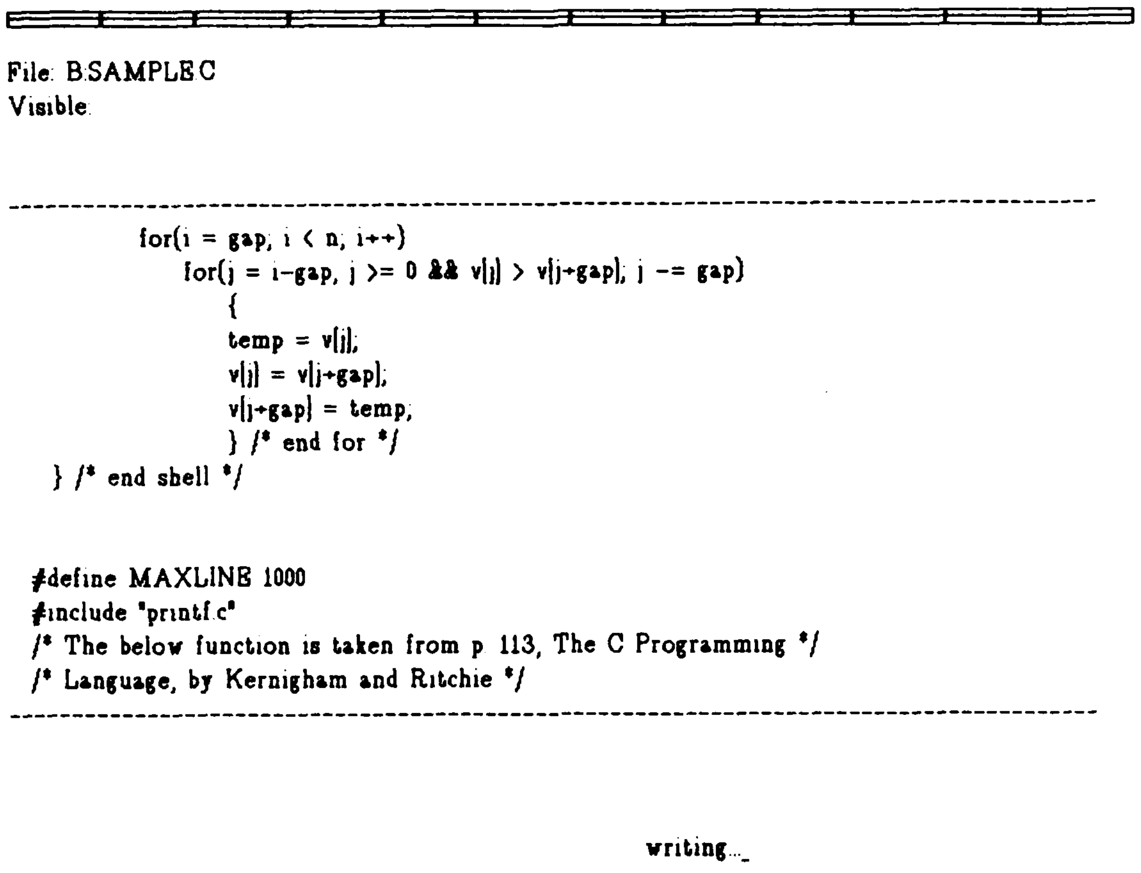
template or free input options. The user selects the template operation and, subsequently, selects the comment template from the preprocessor menu. In Figure 30, the user has continued his comment input beyond one physical line and the editor has wrapped the input to a new physical line.



```
File: B:SAMPLE.C
Visible:        atoi          exit()        fclose(       fopen(        fprintf(
fscanf(         getchar()     itoa(         printf(       putchar(      read(
scanf(          sizeof        sprintf(     sscanf(       strcpy(       write(
--------------------------------------------------------------------------------
        for(gap = n/2; gap > 0; gap /= 2)
            for(i = gap; i < n; i++)
                for(j = i-gap; j >= 0 && v[j] > v[j+gap]; j -= gap)
                    {
                    temp = v[j];
                    v[j] = v[j+gap];
                    v[j+gap] = temp;
                    } /* end for */
    } /* end shell */


  #define MAXLINE 1000
  #include "printf.c"
  /* The below function is taken from p 113, The C Programming */
--------------------------------------------------------------------------------
continue

/* Language,_
```

Fig. 30 C-Smart Screen 16

Figure 31 exemplifies the manner in which the editor advises the user of buffer status. In this example, the user has elected the "save" option from the top level menu. The status line in the user input area provides the



```
File: B:SAMPLE.C
Visible:

--------------------------------------------------------------------------------
            for(i = gap; i < n; i++)
                for(j = i-gap; j >= 0 && v[j] > v[j+gap]; j -= gap)
                    {
                    temp = v[j];
                    v[j] = v[j+gap];
                    v[j+gap] = temp;
                    } /* end for */
    } /* end shell */


  #define MAXLINE 1000
  #include "printf.c"
  /* The below function is taken from p 113, The C Programming */
  /* Language, by Kernigham and Ritchie */
--------------------------------------------------------------------------------


                                                        writing..._
```

Fig. 31 C-Smart Screen 17

"writing..." message as the compiler suitable file is being generated. The "swapping..." message may also appear when the editor needs to page in a new block of program text from the disk. The final status message, "merging..." is displayed when the internal edit buffer has been filled and must be merged with the existing file.

VI. CONCLUSIONS

The primary conclusion to be drawn from this effort is that a basic syntax directed editor can be implemented on a minimally resourced machine. However, space restrictions must be considered when making any significant decision. Time constraints proved to be less of a factor than was originally anticipated. Despite a very slow CPU speed, the total hardware/software system proved able to provide timely responses in most instances. The only operations which were degraded by the low speed concerned the file system. Paging blocks in and out as well as merging files when the edit buffer filled proved to be very expensive. A faster clock would have helped this situation, but could not have completely eliminated the associated user inconvenience.

The space constraints of the project required tradeoff decisions to be made constantly. Many decisions determining the inclusion of facilities were based solely on the availability of space. The system now consumes fifty out of an available fifty-eight thousand bytes of main memory. Thus, only eight thousand bytes remain for run time storage requirements. This has not caused any problems in the testing of the editor to date. It is possible that a program requiring many levels of nested constructs could need more run time memory than is available since the editor accommodates nested structures by recursive procedure calls. Test constructs have been successfully nested to sixteen levels, which is the deepest level that the screen can display and that the compiler used for validation can accomodate.

A. TRADEOFFS

There are several features which were not included in this version due to limited space. The first such feature is holophrasting. The data structure which contains user input is a doubly linked list of records. Each record consists of a line of user input and various syntactic information concerning that line. One of the units of information is the tab level of the line. This information was specifically included to ease the implementation of the holophrasting facility. Further, the tab levels of the templates were designated to support holophrasting. Thus, all that is needed to implement the feature is the code to obtain the user's desired level of view and then retrieve the statements at that level. Many of the utility functions in the editor could be used to support the holophrasting feature.

Another feature which was planned but not implemented for similar reasons concerned a search utility. The data structures are designed to allow the easy retrieval of standard syntactic constructs as units. The user would have been able to retrieve all "for" statements as an example. Again, all that is needed to implement this feature is the user interface.

A significant feature which was planned was a small scale version of the Interlisp DWIM facility. Algorithms were developed to allow the detection and correction of spelling errors which involved one or two characters or transpositions. Again, some of the basic utilities in the editor were designed to support this facility. The implementation of the DWIM feature would have required a large amount of memory, primarily to hold a symbol table, which is not now included in the editor. Another problem with the DWIM feature is that it could not be implemented in a regular manner. Many C programs are created as separate modules. These modules are included with others at compilation to produce a complete program. Identifiers can be declared in one module and referenced in another. Thus, the editor could not build a symbol table which contained all legal identifier references without a knowledge of the dynamic nature of the program. Because it could not be applied regularly and because it would require a great deal of storage, the DWIM facility was not implemented.

The editor does not contain a good search utility, block move, block delete, or search and replace options. These could all be included with very little effort. Also, no type of "UNDO" mechanism is available. The delete function as implemented was designed with an undo capability in mind. Nothing is actually removed from the program. Only pointers in the doubly linked list are changed so that the deleted item is skipped over. Thus, a simple undo facility would be supported by an ability to restore pointers. These features were not included due to time constraints on producing a working program. The editor is now functional and was designed and implemented in only three months of part time effort. The creation of over 2,500 line of source code in that period of time was a large effort, possibly too large.

The editor does feature very good support of comment inclusion, both block and single line. As originally designed, in line comments were to appear only at the end of the program line. This convention was really too restrictive and the supporting code was rewritten to allow comments to appear at any place on any line. This decision change proved to be relatively expensive, primarily due to feature interaction within the editor. First, the editor is able to generate much of the punctuation required by the C language. This is an important con-venience for the user due to the frequency with which errors related to omission of punctuation occur. Also, because C does not include any run time diagnostics, such errors can be difficult to detect. To add ending punctuation, the editor first removes any comments that appear on the line. Then, the program part of the line is scanned for punctuation, which is added if not already present. When comments were only allowed at the end of the line, the saved comments could simply be appended to the correctly punctuated program line. When the comment convention was changed, the original input had to be saved so that it could be compared against the program buffer and the comment buffer in order to correctly reassemble the user input. Further, the editor had to be able to make decisions when the original input did not match the content of either buffer (due to added punctuation as an example). Thus, additional coding and storage were required to accommodate the new comment convention.

A second feature which did not interact well with the comment decision concerned physical line continuations. Under the original comment plan, splitting one logical line across two physical line was relatively easy. If the line contained a comment, the split usually occurred within the comment. If no comment was included, then only the line continuation character ("\") had to be positioned in the input line. When the commenting decision changed, the lines had to be scanned on continuation to determine where the break should occur. Further, if the break came within a comment, it was necessary to determine if more code followed the comment on that same line. In this case, not only did the comment have to be closed and continued, but the line continuation character had to be added to the input outside of the comment. These line breaking problems may not appear to be very significant. However, their complexity was hightened due to the macro substitution capability. Because of this feature, as few as two input characters could generate as many as fifty actual characters. In total, the change to the comment decision caused approximately an additional one thousand bytes of code to be generated. This seems to be an expensive feature. However, the editor should not enforce arbitrary conditions and the user should be able to place a comment anywhere he feels one is needed.

A final significant feature which was planned but not implemented was a type of bounds checking for array references. C does not include any run time ability to make such checks. Originally, it was planned to enable the editor to generate source code which could be implanted in a user program to notify him of bounds violations. This feature proved to be very expensive to implement. Because of the ways in which arrays can be declared, a great deal of code would have been necessary to determine the bounds of an array from its static form. Also, implementation of this feature would have been very difficult without including a parsing capability or a symbol table.

One feature was replaced by another. The editor runs on the CP/M operating system, which has some unusual rules governing the allowable characters in a legal file name. Originally, the editor actually parsed the user supplied file names to ensure their conformance to the CP/M rules. Very good error messages were provided in the event of an illegal file name. This feature was replaced by a facility which checks for assignment statements in conditional expressions. This decision was made because the later error is much more subtle and there is no way of automatically checking for it. Since the operating system would eventually notify the user of an incorrect file name the assignment check was deemed to be more valuable.

B. DESIGN ERRORS

The major flaw in this implementation of the editor concerns its file system. During the primary design process, it was determined that the file system would be as simple as possible so that the major efforts could be directed towards the syntax related parts of the editor. This decision proved to be very costly.

Early in the design process, it was recognized that most of the space requirements of the editor would be driven by its syntax directed nature. Thus, there would be very little space available for text buffers. To overcome this, a small scale virtual memory system was planned to facilitate mass input and output of pages of text. Fixed size pages were designed so that each page contained sixty-four records and one record corresponded to one line of text. Because these sizes were fixed, the space available for each line of text had to be fixed at eighty characters, the maximum allowable per line. Thus, each line (i. e. record) occupies the same amount of space whether it holds a single blank or a full eighty characters.

This paging system has become the major limiting factor and the major inconvenience of the editor. Disk storage space was traded for memory space and the ability to exchange pages of text as rapidly as possible. The speed of the paging operation is still very slow and it is conceivable that using character at a time I/O would not produce a great decrement in system performance. If this is the case, it would be a better decision to eliminate the fixed aspect of the buffers and accept the lower performance. The current version of the editor consumes disk space in six thousand byte blocks. In fact, if only a single character is entered as user input, the editor will require six thousand bytes of disk space to store that character. The storage requirement is the same as if sixty-four lines of eighty characters each had been entered. Because of this, the editor will not be able to operate on source programs which exceed approximately one thousand, four hundred lines in length. The typical disk would not be able to hold the intermediate files of larger source programs.

A much greater effort should have been directed to the filing system early in the design phase. The current file system is so simple that it degrades the performance of the editor as a whole. Also, the space required by the utilities needed to implement paging, disk random access, merging of files, and other similar functions consume a large part of the space required by the editor. Approximately eight thousand bytes of compiled code is dedicated to just these utilities.

A second design error concerns the buffers internal to the editor. There are three such buffers. One buffer holds the characters displayed on the screen. The second holds one page of the user's program read in from the disk. The last buffer is dedicated to insertions which the user may include. These three buffers are physically separated. A better design decision would have been to combine the last two buffers into one, logically separated structure.

The screen buffer must keep track of the physical location of each line being displayed. If any displayed line is edited, its location in memory must be known so that an update can occur. If only one buffer had been used (in addition to the screen buffer) simply storing the array index of each line on screen would have satisfied this requirement. Currently, the array name and the array index must be saved for each displayed line. Then separate sequences of code are required to access and update the screen lines, dependent on which array holds the line. The code sequences are very similar; the primary difference being the array name. Much of the "duplicate" code could have been eliminated if only one buffer had been used.

A third design error concerned the choice of the supported language. Many other languages have a much more regular structure and have readily available grammars. Not only did the C language have to be learned, but a large amount of time was devoted to developing a grammar for the language. The error here is that one should have a very thorough knowledge of a language prior to attempting the implementation of a syntax directed editor to support that language. Moreover, the editor itself is written in C. The amount of code used in this implementation could be reduced, simply because different, shorter code sequences could be used to accomplish the same result in many instances.

C. SUMMARY

A syntax directed editor, although a very powerful and convenient tool, can be implemented on a very basic machine. However, the implementation of the tool should not be taken lightly. Early design decisions which seem peripheral to the effort can later become the major limiting factors of the system if not sufficiently considered. This fact evinces the importance of the design phase. As a general rule, a less than complete design will produce a less than desired output.

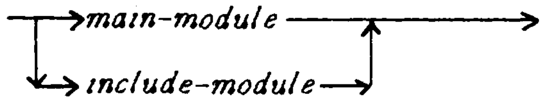
Time constraints on the completion of a project can cause some features to be excluded. An implementation can always be done better in more time. The editor created as a part of this thesis is a useful tool and it can be enjoyable to observe in operation. The system could be much better and could include more facilities had a sufficient amount of time been devoted to its implementation.

APPENDIX A. C LANGUAGE SYNTAX CHARTS

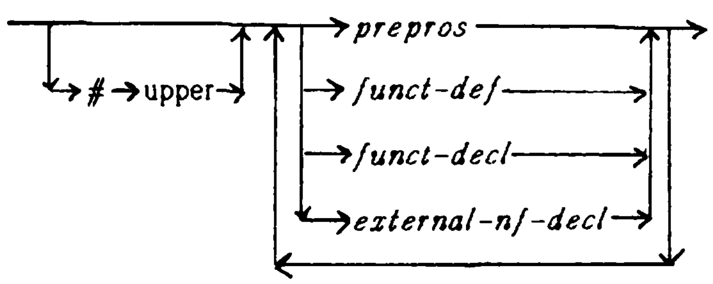
The following charts have been developed primarily from information found in Appendix A to Reference 12. The works of Michael Meissner [Ref. 14] and Patrick Fitzhorn [Ref. 15] have also been consulted.

The syntax charts do not represent an absolute definition of the C language. They have been constructed so as to be as exception free as possible. However, this goal was not always met and the information contained in Appendix A to Reference 12 should be considered as the absolute authority in the event of discrepancies. Certain obvious classes have been omitted from the charts. The class Letter consists of all upper and lower case letters, a thru z. The class Symbol contains any character symbol not represented in the Letter class. 'Character symbol' is meant to exclude digits and the character representation of digits. The *italicized* words in the charts indicate syntactic classes (nonterminals). Letters and words in normal type are meant to indicate verbatim representations (terminals).

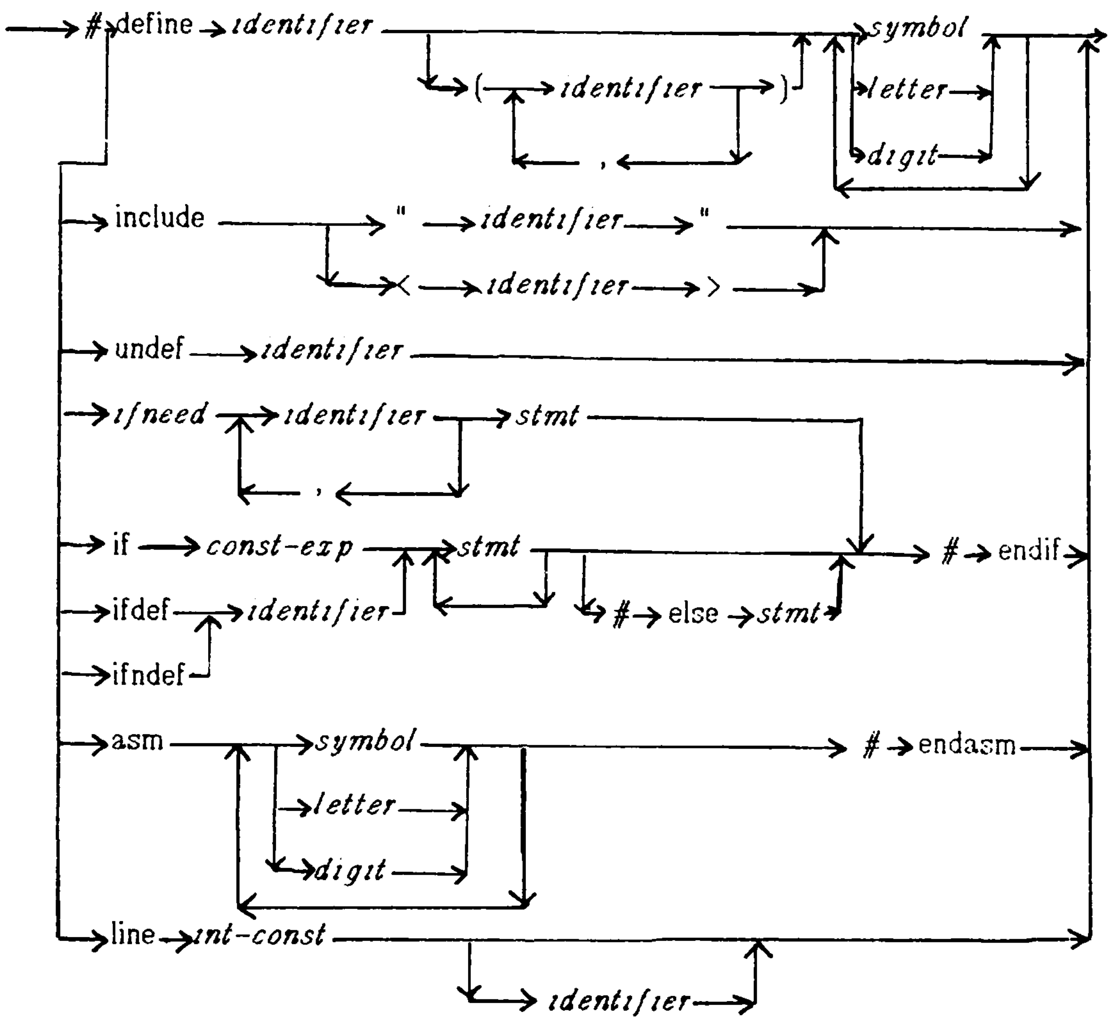
C-program



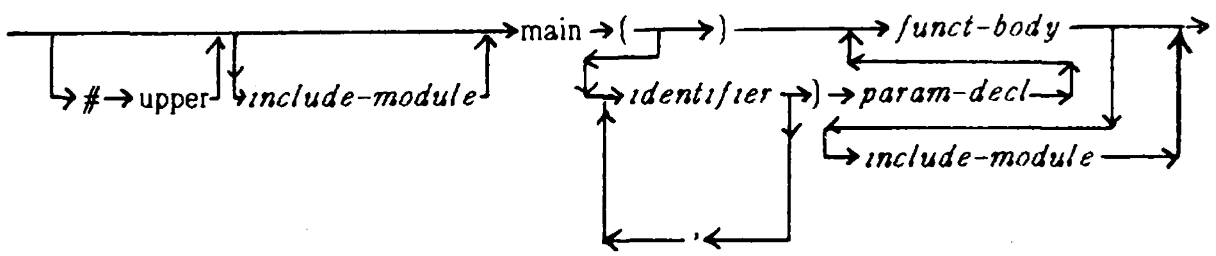
include-module



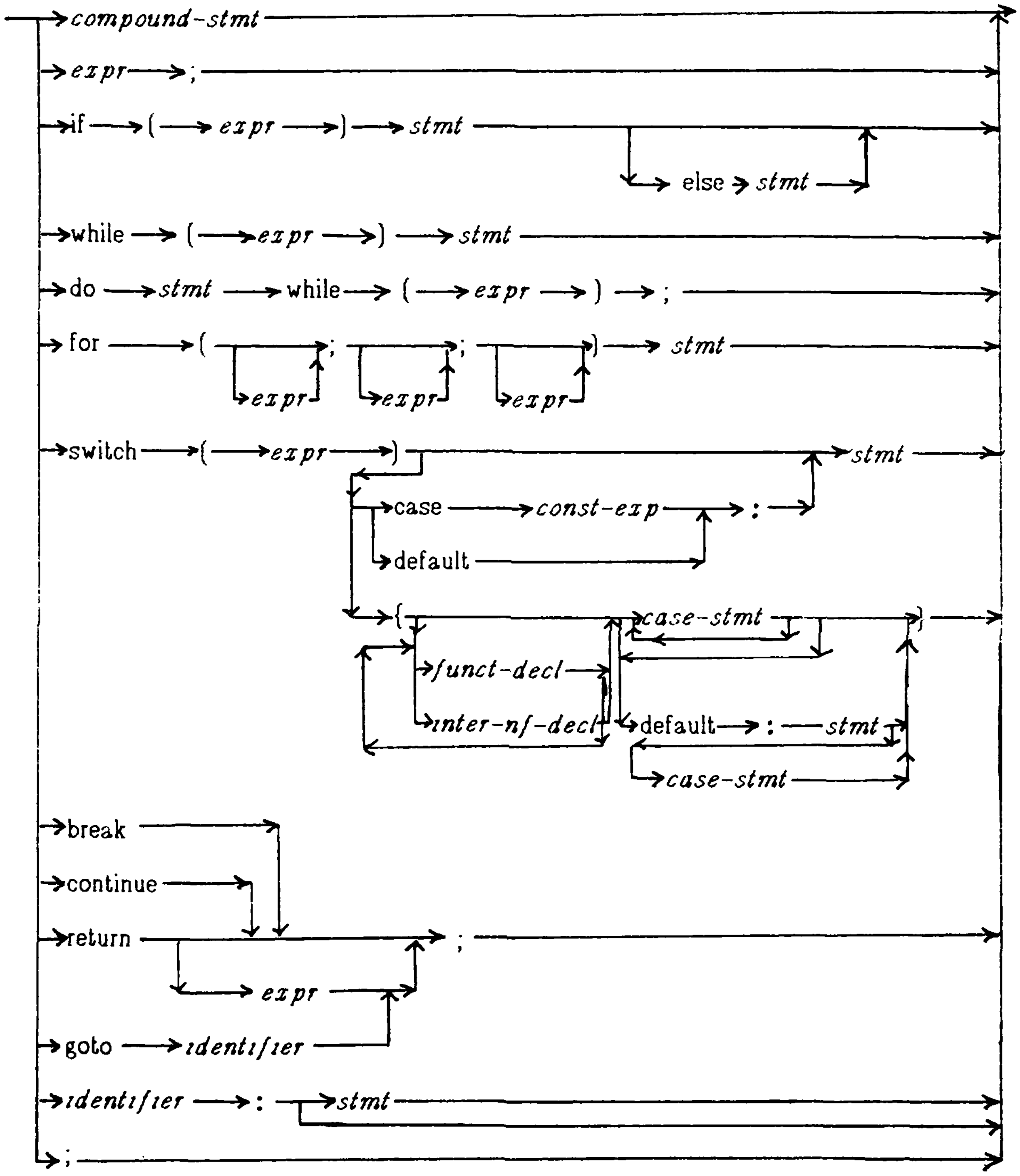
prepros



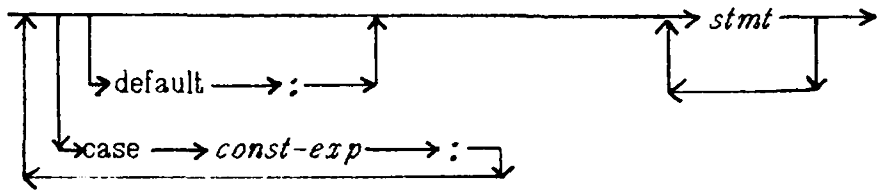
main-module



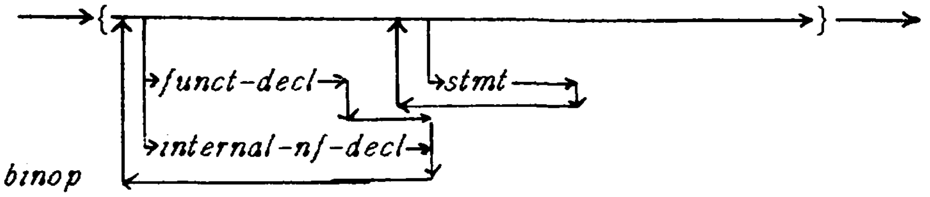
stmt


compound-stmt
expr
;
if
(
expr
)
stmt
else
stmt
while
(
expr
)
stmt
do
stmt
while
(
expr
)
;
for
(
;
;
)
stmt
expr
expr
expr
switch
(
expr
)
stmt
case
const-exp
:
default
{
case-stmt
}
funct-decl
inter-nf-decl
default
:
stmt
case-stmt
break
continue
return
;
expr
goto
identifier
identifier
:
stmt
;

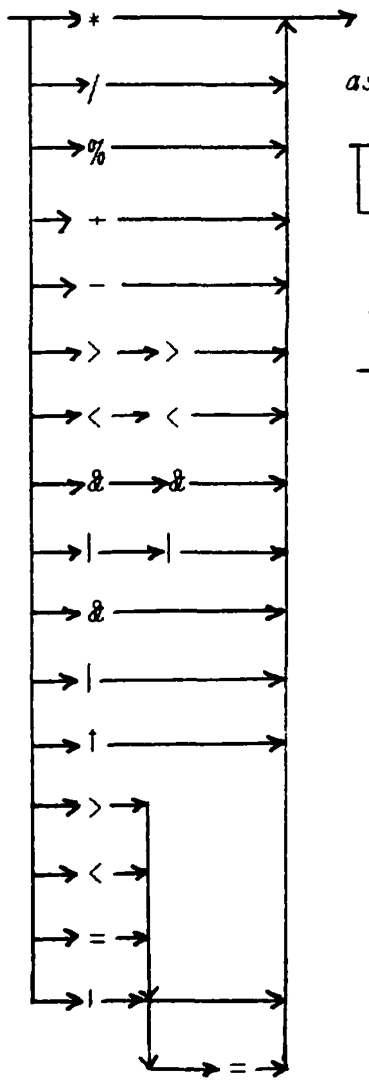
case-stmt



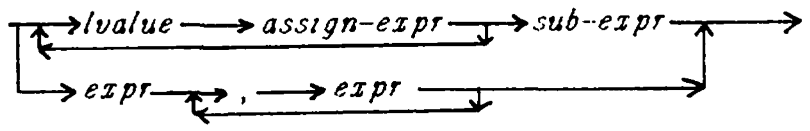
compound-stmt



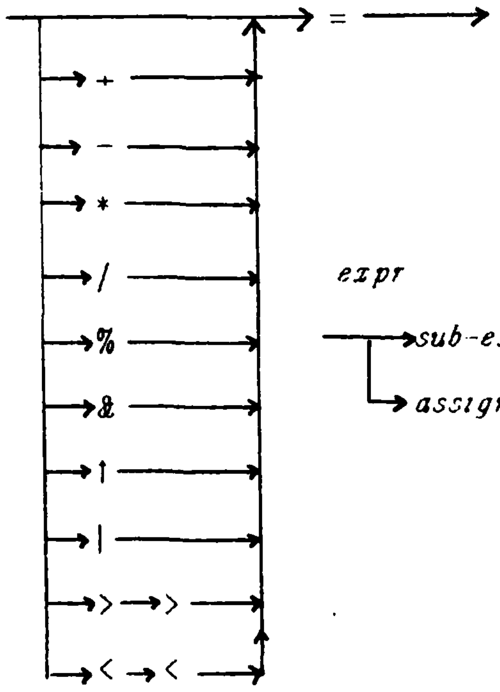
binop



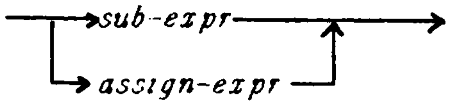
assign-expr



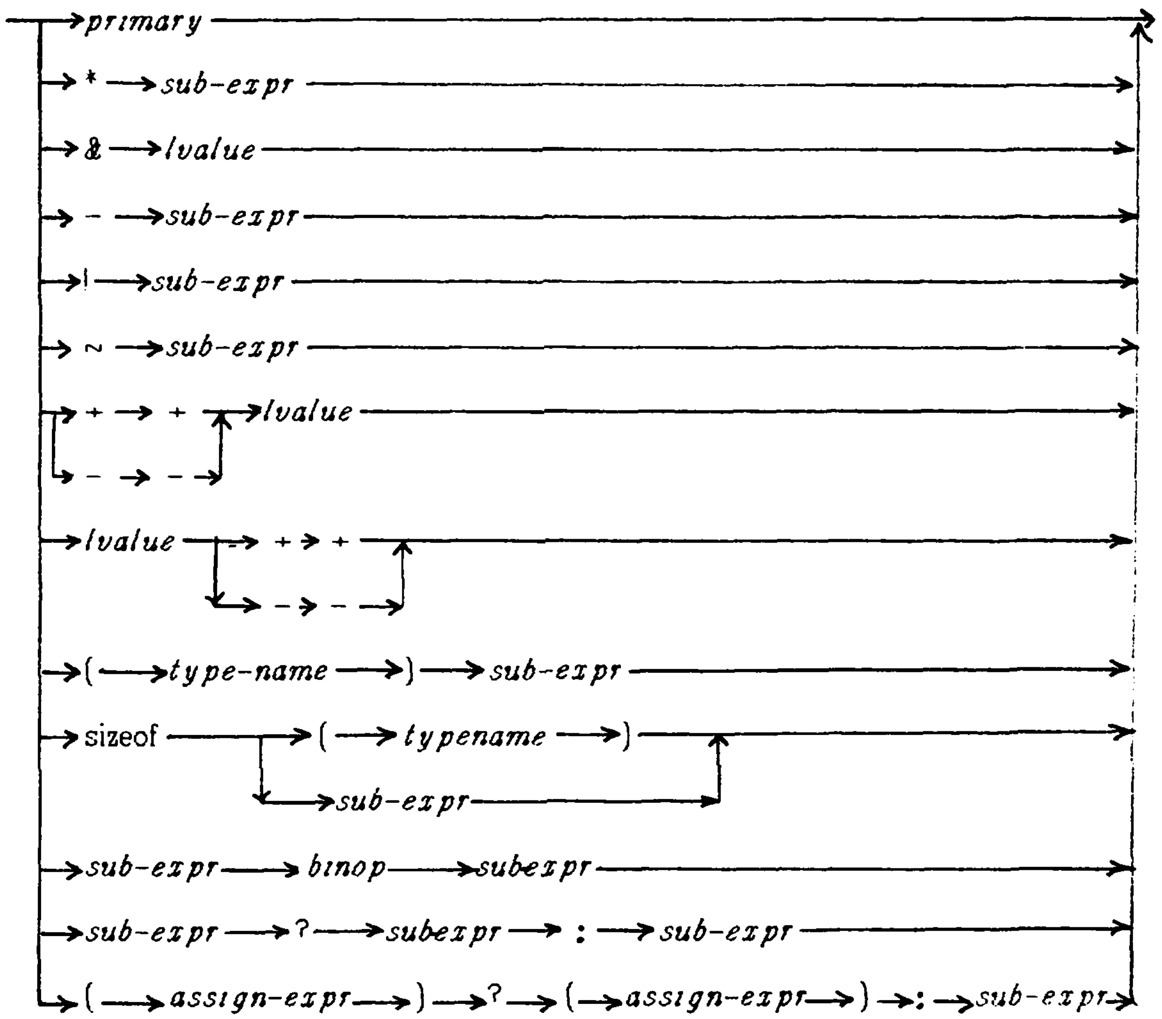
assign-op



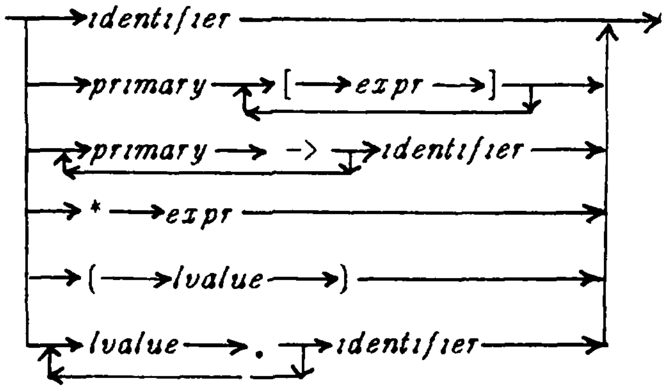
expr



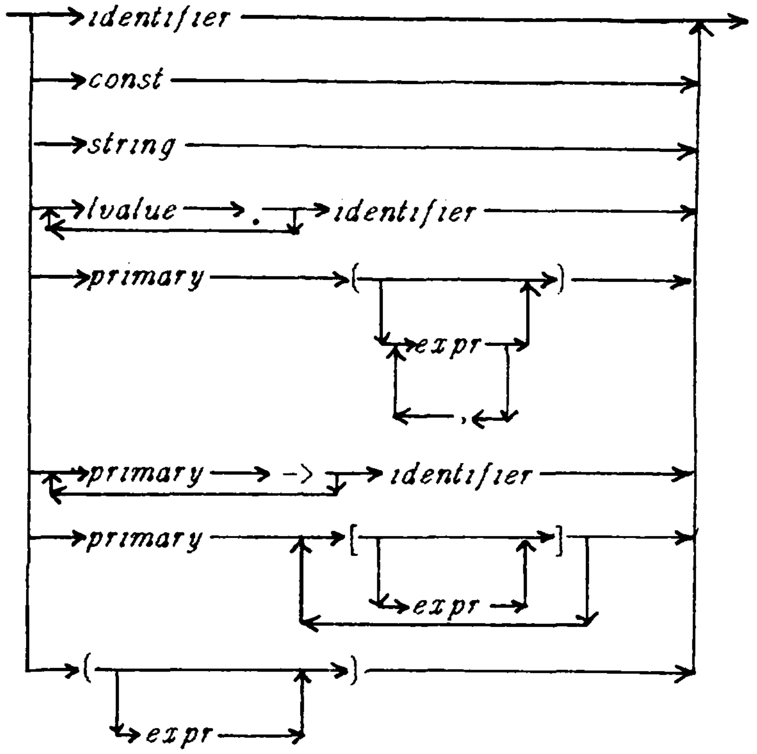
sub-expr



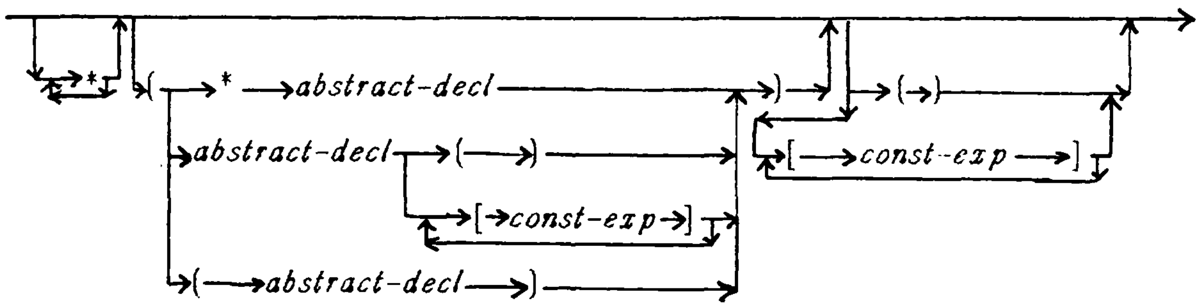
lvalue



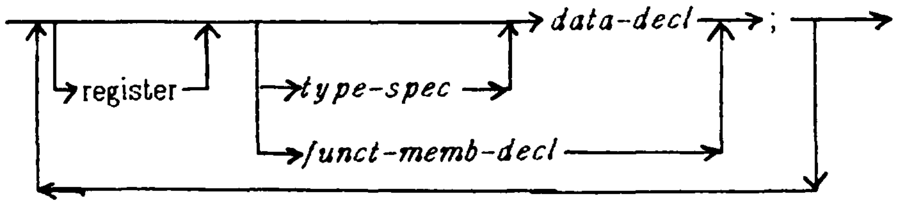
primary



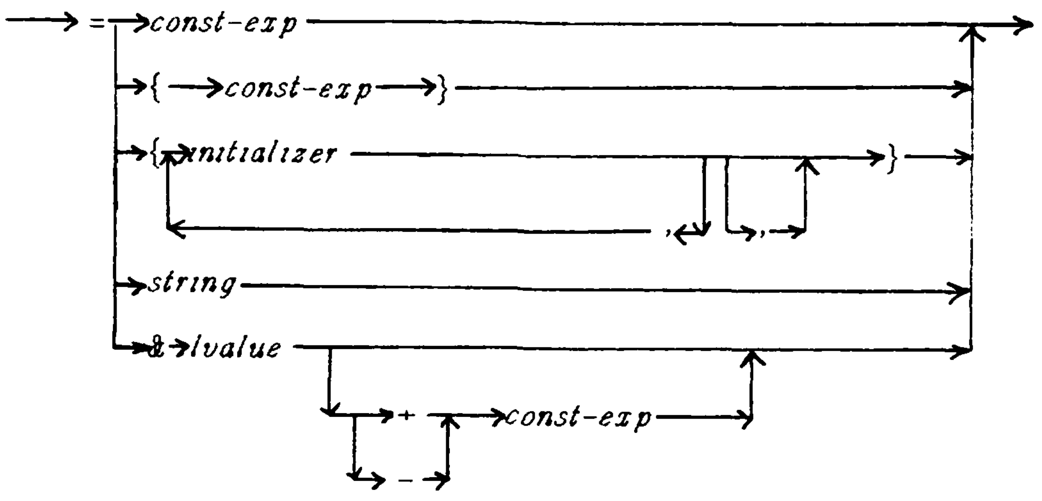
abstract-decl



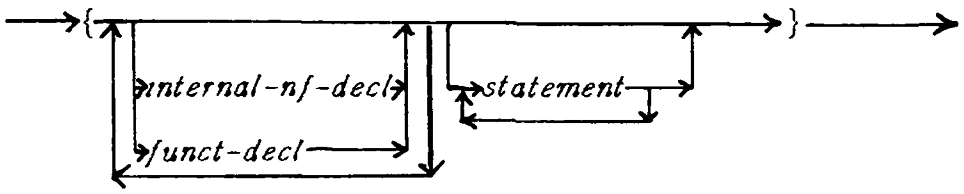
param-decl



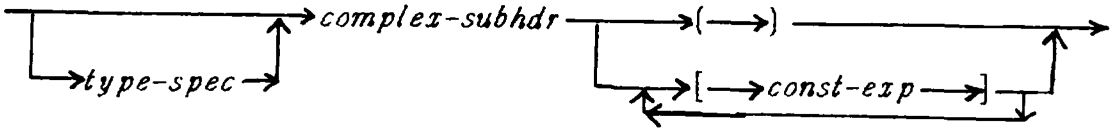
initializer



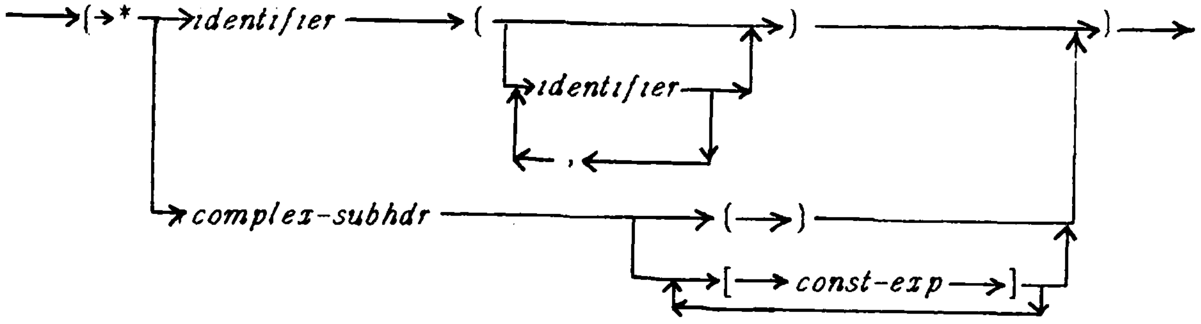
funct-body



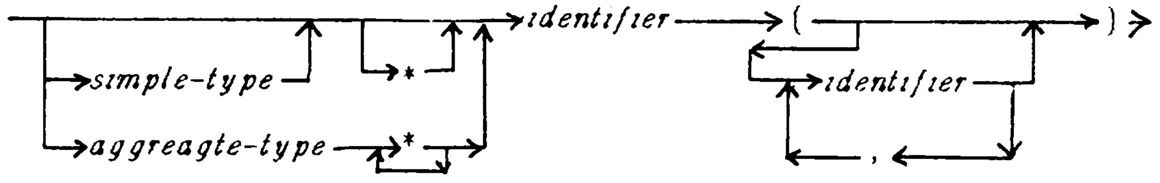
complex-hdr



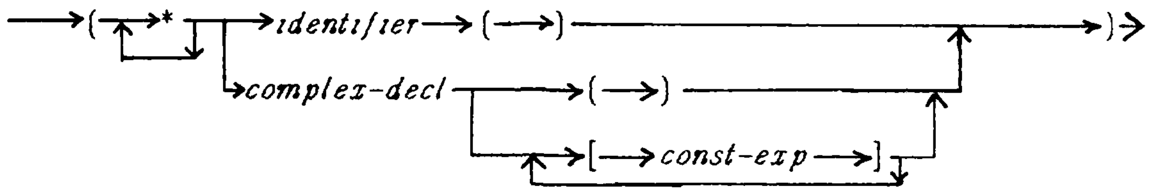
complex-subhdr



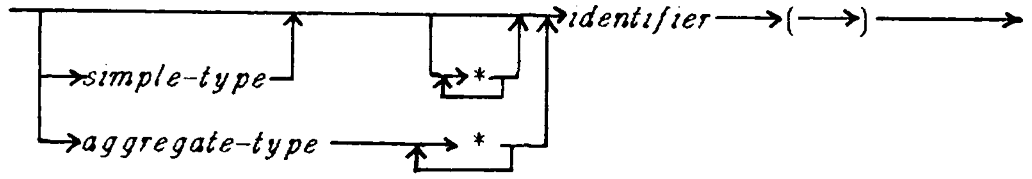
simple-hdr



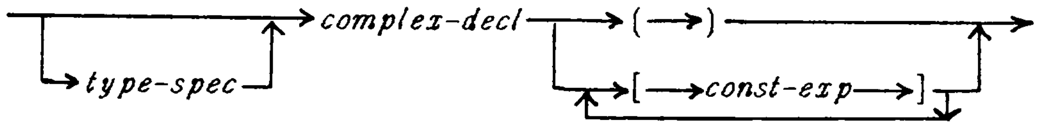
complex-decl



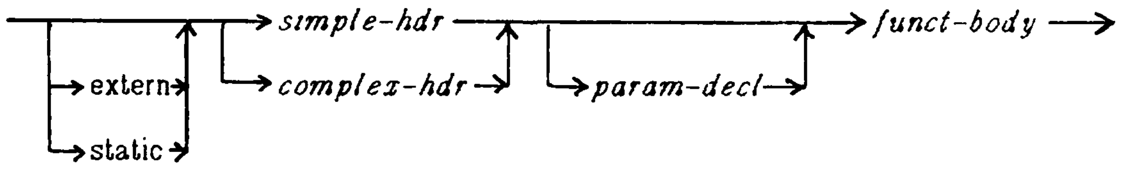
simple-type-decl



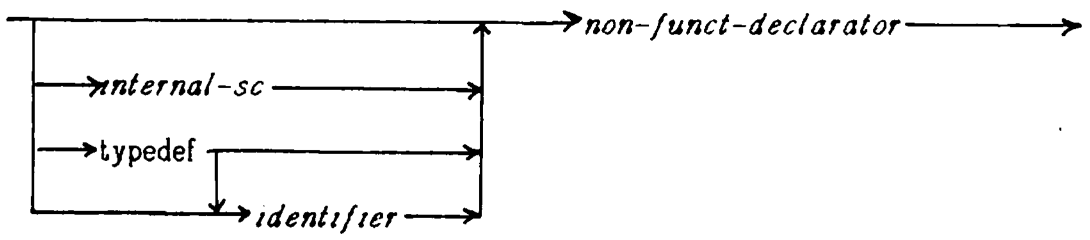
complex-type-decl



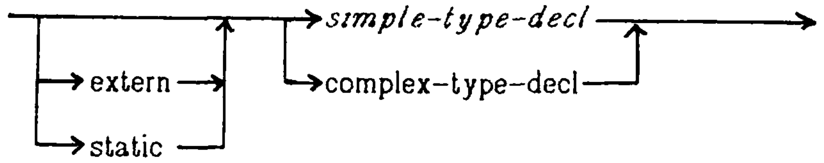
funct-def



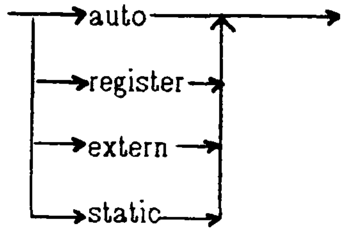
internal-nf-decl



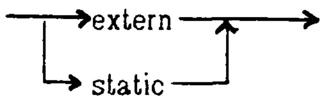
funct-decl



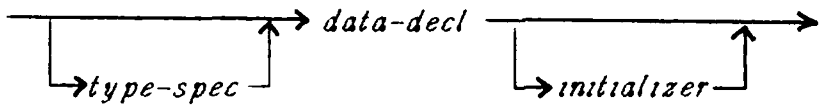
internal-sc



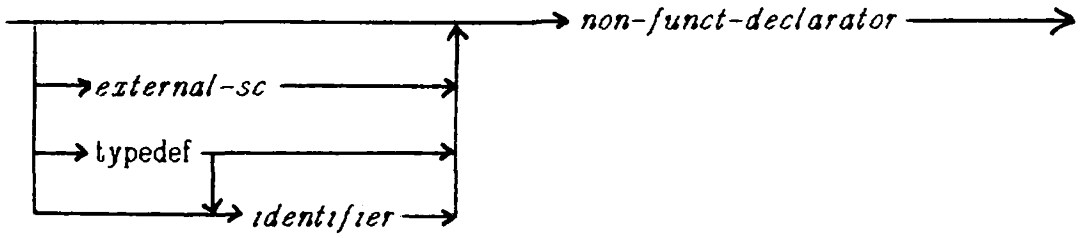
external-sc



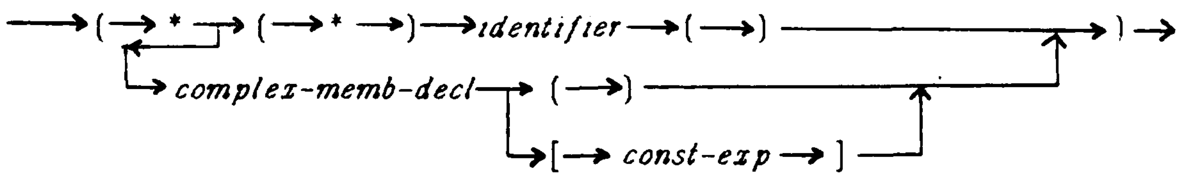
non-funct-declarator



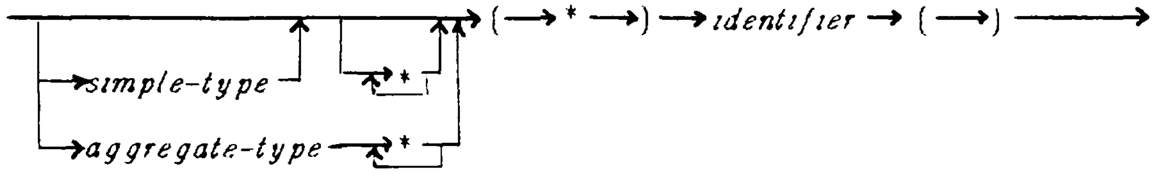
external-nf-decl



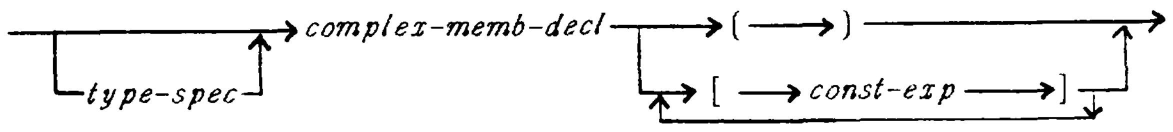
complex-memb-decl



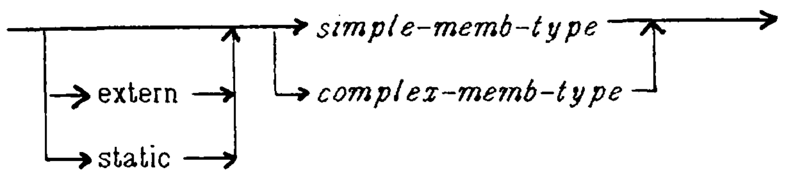
simple-memb-type



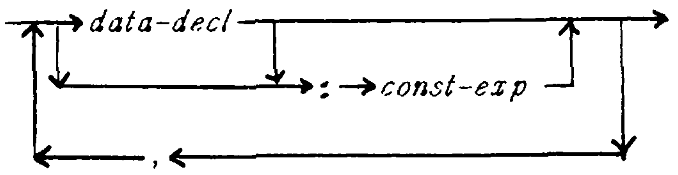
complex-memb-type



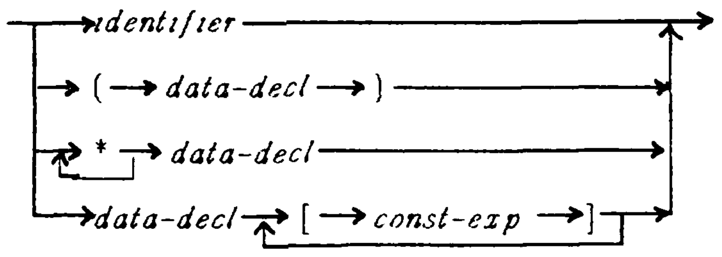
funct-memb-decl



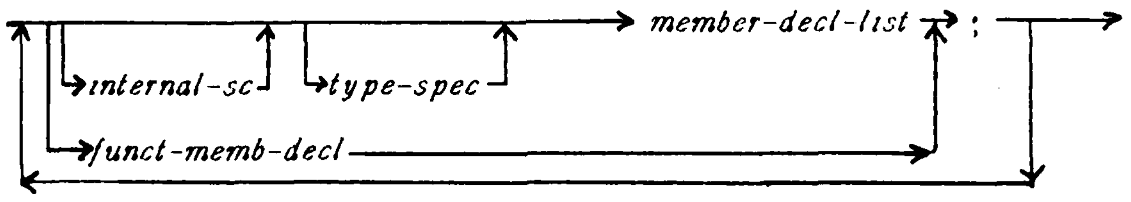
member-decl-list



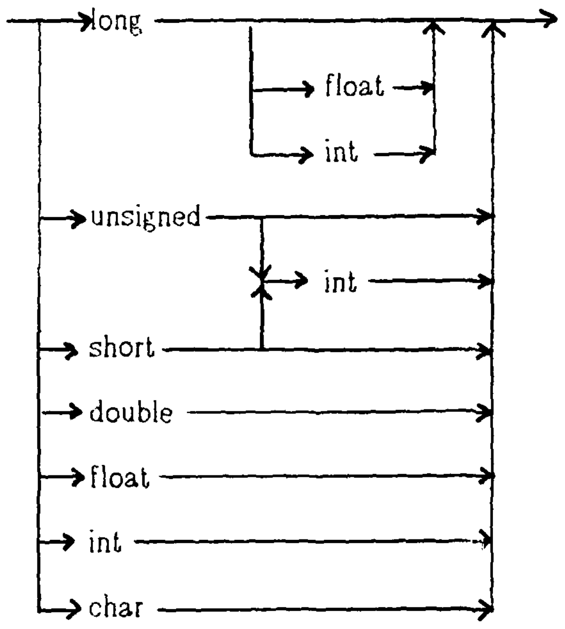
data-decl



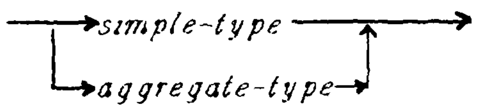
member-list



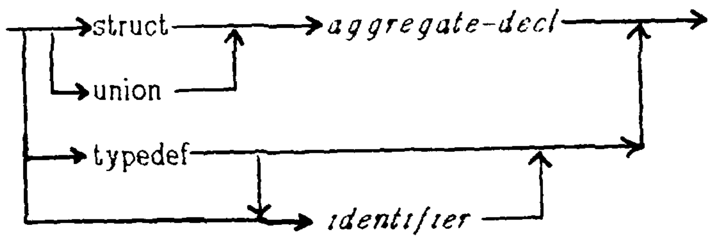
simple-type



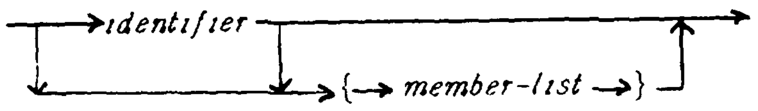
type-spec



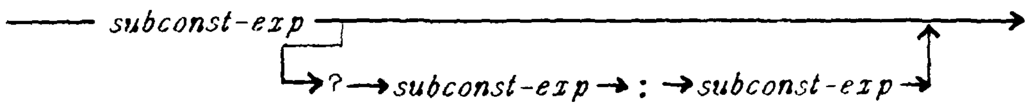
aggregate-type

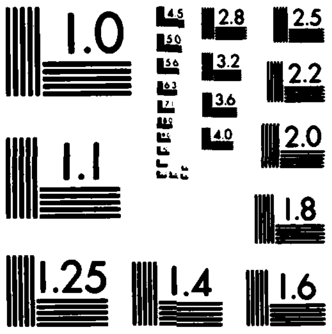


aggregate-decl

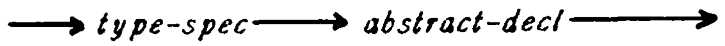


const-exp

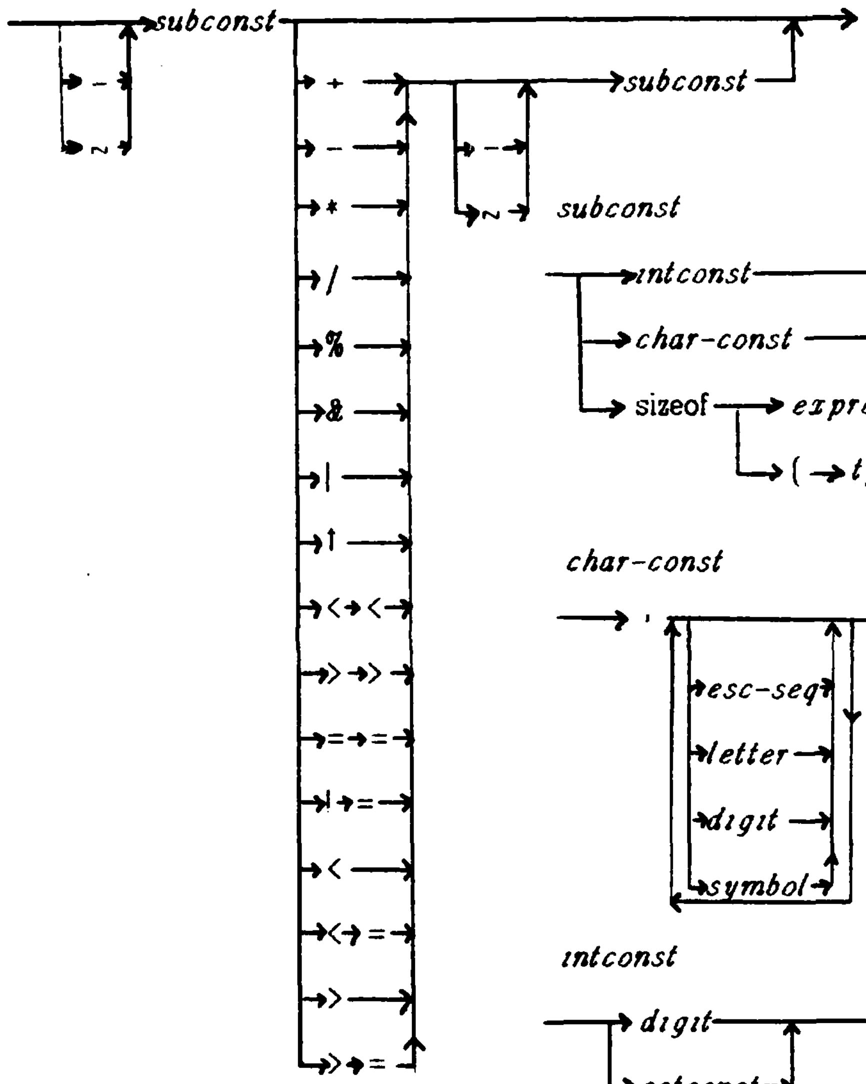



1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

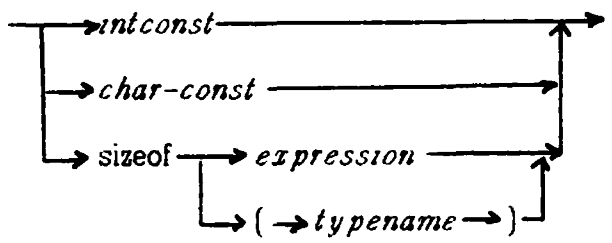
typename



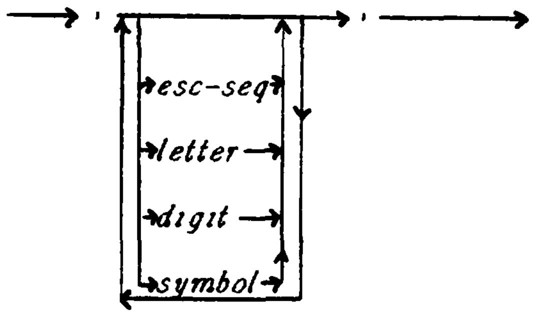
subconst-exp



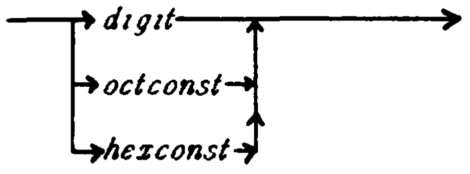
subconst



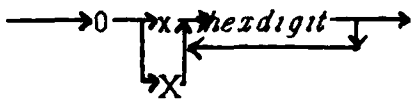
char-const



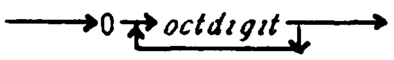
intconst



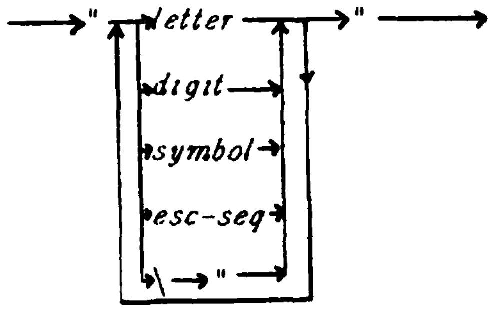
hexconst



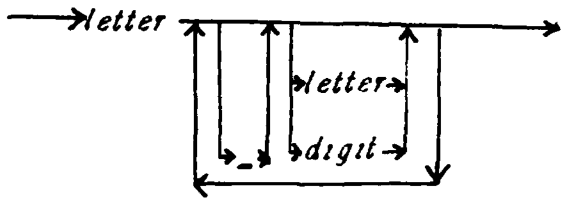
octconst



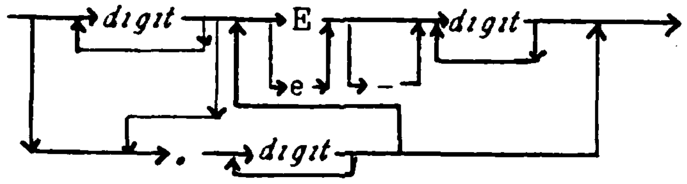
string



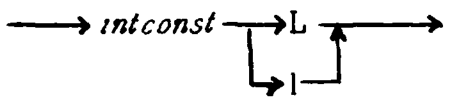
identifier



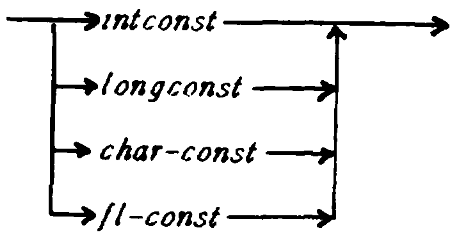
fl-const



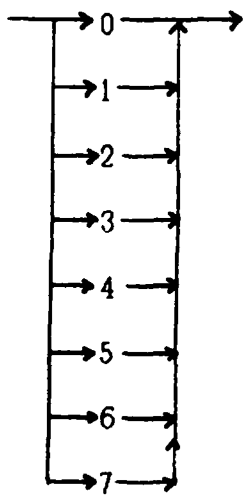
longconst



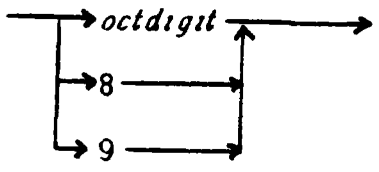
const



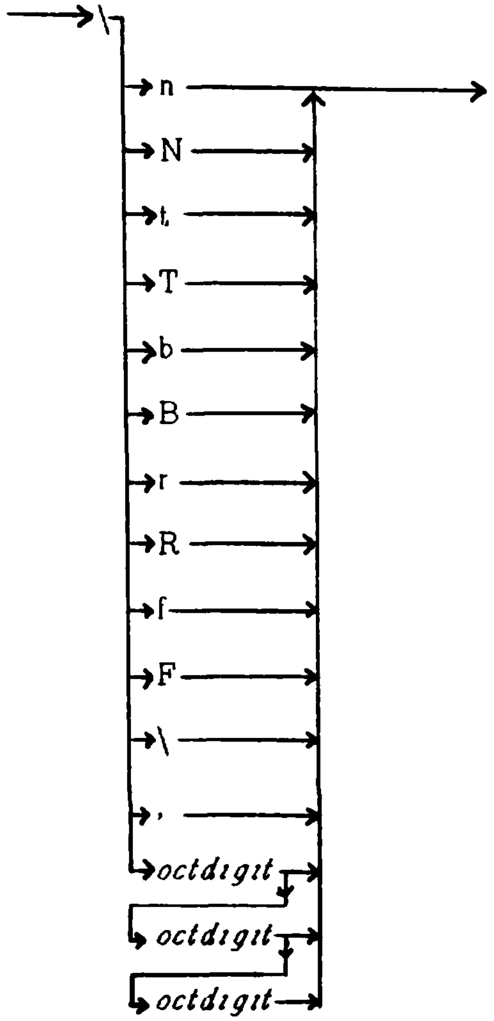
octdigit



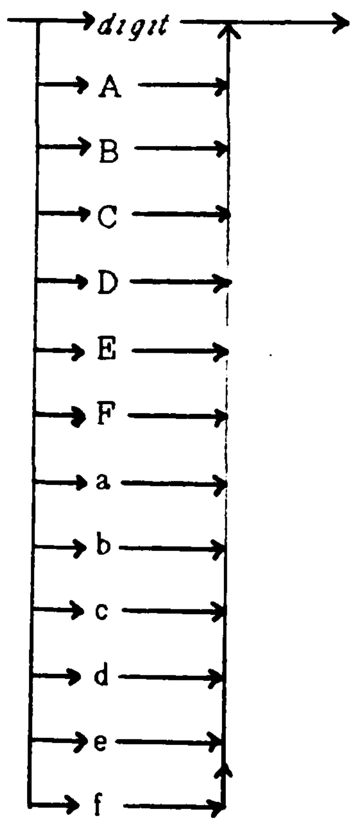
digit



esc-seq



hexdigit



LIST OF REFERENCES

1. Meyrowitz, N. and VanDam, A., "Interactive Editing Systems: Part II", Computing Surveys, v. 14, no. 3, pp. 353 – 397, September 1982.

2. Hansen, W. J., Creation of Hierarchic Text With a Computer Display, Ph. D. Thesis, Stanford University, 1971.

3. Shani, U., "Should Program Editors Not Abandon Text Oriented Commands?", SIGPLAN Notices, v. 18, no. 1, January 1983.

4. Donzeau-Gouge, V., Huet, G. and Lang, B., "Programming Environments Based on Structured Editors: The MENTOR Experience", Interactive Programming Environments, Barstow, D., Sandewall, E. and Shrobe, H., ed., pp. 128 – 140, McGraw-Hill, 1984.

5. Masinter, L. and Teitelman, W., "The Interlisp Programming Environment", Interactive Programming Environments, Barstow, D., Sandewall, E. and Shrobe, H., ed., pp. 83 – 96, McGraw-Hill, 1984.

6. Reps, T. and Teitelbaum, T., "The Cornell Program Synthesizer: A Syantax-Directed Programming Environment", Communications of the ACM, v. 29, no. 9, pp. 563 – 573, September 1981.

7. Wood, S. R., "Z – The 95% Program Editor", ACM SIGOA Newsletter, Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, v. 2, no. 1, pp. 1 – 7, Spring/Summer 1982.

8. Allison, L., "Syntax Directed Program Editing", Software – Practice and Experience, v. 13, no. 5, pp. 453 – 465, May 1983.

9. Conway, R., "COPE, a Cooperative Programming Environment", presented at the Naval Postgraduate School, Monterey, Ca., 28 February, 1984.

10. Ball, E. J., Hayes, P. J. and Reddy, R., "Breaking the Man-Machine Communications Barrier", Tutorial: Software Development Environments, pp. 302 – 313, IEEE Computer Society 1981.

11. Parnas, D. L., "On the Design and Development of Program Families", IEEE Transactions on Software Engineering, v. SE-2, no. 1, pp. 1 – 9, March 1976.

12. Kernighan, B. W. and Ritchie, D. M., The C Programming Language, Prentice-Hall, 1978.

13. MacLennan, B. J., Principles of Programming Languages: Design, Evaluation and Implementation, Holt, Rhinehart and Winston, 1983.

14. Meissner, M., "Correspondence", ACM SIGPLAN Notices, v. 17, no. 8, pp. 84 – 95, August 1982.

15. Fitzhorn, P. A. and Johnson, G. R., "C: Towards a Concise Syntactic Description", ACM SIGPLAN Notices, v. 16, no. 12, pp. 14 – 21, December 1981.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
4.	Assoc. Prof. Bruce J. MacLennan, Code 52ML Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
5.	Prof. Gordon Bradley, Code 52BZ Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
6.	CPT Robert Richbourg, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	2